

**Agent Interaction:  
Abstract Approaches to  
Modelling, Programming and Verifying  
Multi-Agent Systems**

Agent Interactie:  
Abstracte Benaderingen voor  
het Modelleren, Programmeren en Verifiëren van  
Multi-Agent Systemen  
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht  
op gezag van de Rector Magnificus, Prof. Dr. W.H. Gispen, ingevolge het  
besluit van het College voor Promoties in het openbaar te verdedigen  
op maandag 11 november 2002 des middags te 2.30 uur

door

Wietske de Vries

geboren op 12 december 1971,  
te Sneek

promotoren: Prof. Dr. J.-J. Ch. Meyer,  
Instituut voor Informatica en Informatiekunde,  
faculteit Wiskunde en Informatica,  
Universiteit Utrecht

Prof. Dr. J. Treur,  
Afdeling Kunstmatige Intelligentie,  
faculteit Exacte Wetenschappen,  
Vrije Universiteit Amsterdam

copromotoren: Dr. F.S. de Boer,  
Instituut voor Informatica en Informatiekunde,  
faculteit Wiskunde en Informatica,  
Universiteit Utrecht

Dr. W. van der Hoek,  
Instituut voor Informatica en Informatiekunde,  
faculteit Wiskunde en Informatica,  
Universiteit Utrecht

Dr. C.M. Jonker,  
Afdeling Kunstmatige Intelligentie,  
faculteit Exacte Wetenschappen,  
Vrije Universiteit Amsterdam

ISBN 90-393-3197-9



SIKS Dissertation Series No. 2002-14

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

*In dreams begin  
Responsibilities.*

*U2*



---

# Contents

---

<b>Preface</b>	<b>v</b>
<b>1 Introduction: What about Abstraction, Agents and Interaction?</b>	<b>1</b>
<b>2 Abstracting Agent Interaction Histories into Intentionality</b>	<b>19</b>
2.1 Introduction . . . . .	19
2.2 Background . . . . .	23
2.3 Basic assumptions . . . . .	27
2.4 Formal preliminaries . . . . .	29
2.4.1 State Language . . . . .	31
2.4.2 Temporal Language . . . . .	32
2.4.3 Finite variability and change pinpoint principle . . . . .	34
2.5 Criteria for beliefs, desires and intentions . . . . .	37
2.5.1 Beliefs . . . . .	37
2.5.2 Intentions . . . . .	39
2.5.3 Desires . . . . .	43
2.5.4 Changing the BDI-criteria . . . . .	45
2.5.5 Ascription and checking . . . . .	48

2.5.6	Example . . . . .	49
2.6	Anticipatory reasoning in organisations . . . . .	51
2.7	Discussion . . . . .	55

### **3 Reuse and Abstraction in Verification: Agents Interacting with Dynamic Environments 61**

3.1	Introduction . . . . .	61
3.2	The domain of agents in a dynamic world . . . . .	66
3.3	Temporal models and temporal languages . . . . .	71
3.3.1	Basic concepts . . . . .	71
3.3.2	The language abstraction formalism . . . . .	74
3.4	Proving successfulness of actions . . . . .	75
3.4.1	Approaching the problem of proving successfulness of actions . . . . .	76
3.4.2	The system of coordination properties . . . . .	80
3.4.3	The proof of action successfulness . . . . .	83
3.5	Relating abstract notions to detailed meaning . . . . .	84
3.6	Basic properties . . . . .	106
3.6.1	Properties concerning actions starting and ending . . . . .	106
3.6.2	Rationality properties . . . . .	106
3.6.3	Properties concerning facts and observations . . . . .	108
3.6.4	Properties of interaction . . . . .	109
3.6.5	No-overtaking properties . . . . .	111
3.7	Proving COORD0 . . . . .	113
3.7.1	Properties needed for proving COORD0 . . . . .	115
3.7.2	The proof of COORD0 . . . . .	119
3.8	Proving COORD5 . . . . .	131
3.8.1	Properties needed for proving COORD5 . . . . .	132
3.8.2	The proof of COORD5 . . . . .	134
3.9	Verifying an instance of the class: the Mouse . . . . .	136
3.10	Discussion . . . . .	140

### **4 An Abstract Model for Agents Interacting in Real-Time 143**

4.1	Introduction . . . . .	143
4.2	Informal explorations . . . . .	148
4.2.1	Cycles of reasoning and interaction . . . . .	148
4.2.2	Multiple parallel behaviours? . . . . .	152
4.2.3	Group action and individual action . . . . .	154

4.3	Syntax . . . . .	158
4.3.1	Basic sets . . . . .	159
4.3.2	Auxiliary languages . . . . .	160
4.3.3	Syntax of the programming language itself . . . . .	169
4.4	Semantics . . . . .	171
4.4.1	Intuitions . . . . .	171
4.4.2	Local semantics: agent traces . . . . .	176
4.4.3	Global semantics: system traces . . . . .	186
4.5	Proof of correct reason–interact behaviour . . . . .	193
4.6	Illustration . . . . .	209
4.7	Conclusions . . . . .	212
<b>5</b>	<b>An Abstract Programming Language for Agents Interacting through Group Actions</b>	<b>217</b>
5.1	Introduction . . . . .	217
5.2	Intuitions . . . . .	220
5.3	Syntax . . . . .	226
5.3.1	Basic sets and conventions . . . . .	226
5.3.2	Syntax of programs . . . . .	230
5.4	Semantics . . . . .	235
5.4.1	Preliminaries . . . . .	236
5.4.2	Local semantics . . . . .	238
5.4.3	Global semantics . . . . .	244
5.5	Illustrations . . . . .	254
5.6	Conclusions . . . . .	263
<b>6</b>	<b>An Abstract Coordination Language for Agents Interacting in Distributed Plan–Execute Cycles</b>	<b>265</b>
6.1	Introduction . . . . .	265
6.2	Plan features and constraint stores . . . . .	269
6.3	Syntax and semantics of the plan coordination language . . . . .	274
6.4	Illustration . . . . .	286
6.5	The coordination architecture . . . . .	294
6.6	Conclusion . . . . .	296
<b>7</b>	<b>Conclusion: Abstract Interaction is what Agents are for!</b>	<b>301</b>
	<b>Bibliography</b>	<b>313</b>

<b>Samenvatting</b>	<b>327</b>
<b>Curriculum Vitae</b>	<b>331</b>

---

# Preface

---

It is great to be out in the sun.

To me, working on this thesis has been a puzzling, fascinating and enlightening experience. Even now, when all is about to be finished, I still can't entirely figure out my reasons for starting and staying in research, though I am very glad that in the end I did stay (which for a large part is due to the support of others; read on ...). Research itself has also been puzzling. Often, I have felt like a child lost in the woods on a moonless night, looking for the way home. Science brings compelling riddles to be solved, that can sometimes torture the mind by their mere existence. At the end of many days of thinking and typing at the University, I only seemed to have discovered more questions and discarded previous answers, and came home with a brain so tired that watching television (the dumb programs) was the only attractive evening pursuit.

But still, in spite of the puzzlement, it has been fascinating to see how new concepts arise from faint ideas and hazy intuitions, slowly growing into mature scientific results when nurtured well. The charms of logic and the magic of words in the end often showed a path out of the forest and into the sunlight. I have greatly enjoyed the creative experience of making up new universes of agent concepts, even though the struggle to achieve this creation often seemed too long. But the darkest part of every night is just before the dawn, and experiencing a sunrise is always overwhelming.

I would like to thank everyone who supported me through the past six years. First, I thank Jan Treur, for allowing me to share his clear vision on agents, and for letting me go when we found out that our visions were too far apart. Catholijn Jonker has always been very supportive and encouraging, and I owe her a lot. I thank John-Jules Meyer for his enthusiasm and his inspiration, and of course for the opportunity to finish my PhD in Utrecht. He always made me believe that my ideas in fact were going somewhere, even if I didn't see it yet. I thank Wiebe van der Hoek for always exactly understanding what I meant, even when things were still so vague that I lost track of my own thoughts, and for his careful attention for all the tiny technical details as well as for my personal well-being. Frank de Boer I thank for sometimes being there with great ideas and for sometimes being away, allowing me to go my own way.

Apart from my supervisors, there are many others I am grateful to. I thank the members of the reading committee, Michael Wooldridge, David Kinny, Joost Kok, Frank van Harmelen and Rineke Verbrugge, for their attentive reading. I thank my former colleagues at the Vrije Universiteit, especially Frank Cornelissen (it was nice to share a room with someone who lives on music as well), Niek Wijngaards, Pascal van Eck, Pieter van Langen, Annette ten Teije, Frank van Harmelen, Lourens van der Meij, Wouter Wijngaards, Frances Brazier, Joeri Engelfriet and Rineke Verbrugge. At the Utrecht University, I thank Paul Harrenstein (logic games might indeed be fun, though you could never quite convince me), Rogier van Eijk, Marco Wiering, Henry Prakken, Mehdi Dastani, Richard Starmans, Gerard Vreeswijk and Frank Dignum. It has been a pleasure working alongside you all. Koen Hindriks and I never quite agreed, but our discussions always kept me on edge, for which I am truly grateful. I thank Virginia Dignum for enduring and perhaps even enjoying my out of key singing along with the radio, as well as everyone in the vicinity of our office who never seemed to be bothered by this mild form of auditory harassment.

And then there are friends and family. It has always been extremely good to know that there are people you can come home to, and who believe in what you do, even if they don't understand anything of it. I thank Marieke and Jeroen and their two lovely kids Kahdra and Kilian, for their friendship and for allowing me to crash into their home whenever I felt like it. When the world seems to be filled with only formulas, it helps a lot to be playing on the floor with two marvellous little children. Berthy has always been my sister in science (she got her Law PhD degree last March); it was very good to have a friend going through similar struggles. I also thank Hester, Ewald, Petra, Koos and Annelies. Ellen kept me going with her postcards and her being so proud of me; thanks! My dear friend Gerrit continuously dragged me along to parties and other joyous gatherings involving lots of dancing, which was very beneficial. I

thank him and Peter for agreeing to be my two lovely paranymphs. I am very glad to thank my parents, my brothers and sisters and my aunt Wietske for their unconditional love and support. More specifically, I thank Jantina for allowing me to use her laptop, and for aiding with the cover design. Jan-Ybo also had a big explanatory role in the cover; thanks. I thank Martha and Matthijs for always listening to my long expositions on the state of progress of this book, and for the evenings we spent playing games. And I thank Gabriël for being his much appreciated funny and kind self.

Most of all, I am truly grateful to the One who has created the universe, including science, the deep dark forests, my mind, creativity and the sunlight. Without Him, everything would have been meaningless.

*Wieke de Vries,  
Utrecht, the first of October 2002.*



# CHAPTER 1

---

## Introduction: What about Abstraction, Agents and Interaction?

---

*Nothing worth having comes without some kind of fight  
Gotta kick at the darkness till it bleeds daylight*

*Bruce Cockburn*

At first sight, the title of this thesis seems to contain a lot of vague yet fashionable terms used in modern computer science and in artificial intelligence in particular. We live in times where many web pages are ‘interactive’. In the near future, companies may advertise their systems with the slogan ‘agents inside’, while at present the agent research community is still debating and outlining the exact nature of agents. The phrase ‘abstraction’ in science has so many uses that, without further details, it doesn’t have clear intuitive connotations. And the common sense meaning of ‘abstract’ isn’t very encouraging either, as ‘abstract’ is used in the sense of ‘hazy’, ‘not well elaborated’ or even ‘too inspired to make sense’ (when referring to paintings). In this thesis, we will nevertheless use these phrases often to discuss useful features of the software paradigm of multi-agent systems. We start by clarifying our use of the phrases ‘abstraction’, ‘agents’ and ‘interaction’, and by explaining why abstract approaches to agent interaction are beneficial.

## Abstraction

In the relatively brief history of computer science, abstraction has been one of the driving forces of the development of programming languages and software engineering paradigms. With abstraction, we mean the process of defining concepts to describe, direct or analyse the computation processes taking place in computer systems, reducing complexity by leaving out details. A computer system can consist of several connected computers, each of which is a complex piece of hardware, with one or more processors, memory chips, a central clock which sets the pace of the system and connections to transfer bitstrings from one part of the system to another. Fortunately, it isn't necessary anymore to understand the hardware of computers in order to program them. Programming languages have been developed, which abstract from electrical details and allow the programmer to view the computer as an information processor. Compilers and interpreters translate computer programs to lower-level languages, which are close to the machine level and can be directly processed by the hardware implementation.

Different programming paradigms use different concepts to model the computation, storage and information passing of a computer system. We will review several software paradigms. We mainly focus at command-based programming approaches (which are programming languages where the basic operations are commands and programs basically describe sequences of commands), although we will also briefly discuss other paradigms, such as functional programming, logic programming, and constraint programming. In this overview, we discuss the use of abstraction for structuring *data* and *control* in computer programs. We refer to [51, 112, 120] for more extensive descriptions of the different paradigms. We start at the lowest level of abstraction above the binary machine languages.

Here, we have the *assembly languages*. These languages are very close to the machine level. The model of computation is performing simple operations on values in memory locations, such as shifting data from one memory slot to another, or adding two values. Variables are used to represent memory locations. Assembly languages abstract from the binary nature of computer representations and provide the programmer with the possibility to use mnemonic terms to refer to memory locations and program instructions. The only kind of data values are integers; other data types are simulated in terms of these. The contents of memory locations can be tested, using relations like  $<$  and  $=$ . Depending on the outcome of these tests, the control can jump to alternative program lines. The earliest assembly languages did not facilitate structuring of programs in any way; there was no mechanism to group together

lines of code that implement one particular operation and name them to indicate their function. Later on, assembly languages improved in this respect: named subprograms (macros) are introduced, such that a group of statements can be considered as a single action. But it remains possible to jump to another program statement from every point in such a subprogram. The low-level data representation, together with the jump-wise nature of the control, makes assembly programs very hard to read, write and debug. A program mainly is an amorphous heap of manipulations of memory locations. It is very hard to discover whether an assembly program is a game of pacman or a library filing system, as assembly gives little possibilities to include the intuitions of the programmer in the representation of the program. Information passing is also done in a rather primitive manner, through shared variables. When a value created by one part of the program is needed in another part, the program parts use the same variable to store and reference the value.

The paradigm of *imperative programming* is a large step up in abstraction level. Imperative languages have a richer vocabulary than assembly languages; they contain more different statements and control constructs. In general, in order to create the effect of a single statement in an imperative language, several assembly statements would be needed. Thus, programs can be shorter and more understandable. Imperative programming languages provide facilities to structure the program into a main program and several procedures and/or functions. Procedures and functions are named pieces of a program. The names (when well-chosen) can provide intuitions about the meaning of procedures and functions. Control moves from one program location to another through procedure calls. Jump-like statements do not occur anymore in most imperative programming languages (except for the infamous GOTO-statement [38]). These features enable the programmer to write neatly structured programs, where each procedure has a conceptually independent function. Like in assembler, variables are used to represent values. In imperative programs, values can be of different data types, such as characters, integers, booleans, or reals (real numbers). Imperative languages also provide structured data types, for example arrays and sets, such that the data representation chosen can provide intuitions about the relations between values. The user has the possibility to define his or her own data types, starting from the built-in types and type constructors. Data can be made local to a procedure, hiding it from the rest of the program. A procedure can take a number of parameters, which are special variables that implement the transfer of information to procedures and back. Actual values or variables are passed to the procedure when it is called, and when the procedure finishes, values or variables are passed back to the location from which the procedure was called. Procedures provide process abstraction, as each pro-

cedure implements some higher-level operation, such as sorting an array, and it is not necessary to always keep in mind the details of the code of the procedure when using it to sort a particular array. Information passing is done using the parameter mechanism of procedures. Additionally, global variables (variables visible to the whole program) can be used, though this is sometimes considered bad practice, as using global variables overrides the separation of data into logical units belonging to procedures. In short, imperative languages provide structure, for data as well as control.

The next programming paradigm we consider is *object-oriented programming*. In this paradigm, computation and data are encapsulated in units called objects. While in imperative programs, data is local to program parts, in object-oriented programming, the processes that operate on the data are also localised. Data units which conceptually belong together and the functions which operate on this data, are grouped together in an object. Implementation details of both data and operations on the data are hidden inside the object and not accessible to its environment (the other objects and the user). Functions (or procedures) are called methods in this paradigm. For example, an object can represent a library. The object has data structures for the administration of the library collection (which books have been lent, where each book is located) and for the personal data of the members of the library. The library object can have methods for borrowing a book, or for registering a new member. The only way to access the data encapsulated in an object is by using the methods of the object. Methods are parameterised, as are procedures in imperative programs. Objects pass values around through method invocation. Compared to imperative programming, object-oriented programs provide still more structure, as processes are also localised into objects. There are several object classes, which are generic stencils, that act as an object factory. New objects can be created during execution by generating an instance of a class. The object classes form a hierarchy, where the top classes are very generic, and the classes in lower branches of the hierarchy are tailored to a specific function. All features of a class (variables and methods) are inherited by child classes. Thus, the object-oriented paradigm provides facilities to reuse objects and classes for different applications. As all data is encapsulated in objects, information passing takes place through method invocation and the associated parameter mechanism. The object-oriented paradigm uses many features of the imperative paradigm, and adds new concepts, such as data encapsulation. Nevertheless, it is an independent paradigm, as the object-oriented view on computing is entirely different from the imperative view. In object-oriented programming, control is centred in the objects, which means that the responsibility for the computations is inherently more distributed.

The last abstraction step up we take here is the step towards *agents*. Like in objects, agents encapsulate both data and processing of data. The most important innovation of the agent-oriented paradigm is the use of anthropomorphic concepts. Instead of using variables and values as basic concepts, as the previous paradigms did, agents process beliefs, goals, plans, desires, intentions, perform actions and observations, and communicate requests and information to other agents. As these concepts have an intuitive common-sense meaning, they provide a powerful abstraction from details of their particular implementation, which can be helpful in the design and construction of complex software. We repeat the words of Hofstadter here, written in 1981 [64]:

“But eventually, when you put enough feelingless calculations together in a huge coordinated organization, you’ll get something that has properties on another level. You can see it – in fact you *have* to see it– not as a bunch of little calculations, but as a system of tendencies and desires and beliefs and so on. When things get complicated enough, you’re forced to change your level of description. To some extent that’s already happening, which is why we use words such as ‘want’, ‘think’, ‘try’, and ‘hope’, to describe chess programs and other attempts at mechanical thought. Dennett calls that kind of level switch by the observer ‘adopting the intentional stance’.”

The computation model of agent-oriented programming is that of multiple agents, situated in an environment, which they change by performing actions and about which they obtain information by performing observations. Locally, each agent has a mental state where it stores its beliefs, goals, plans, desires, or whichever informational and motivational attitudes it employs. This mental state replaces the association of values to variables, which constitutes the state in the earlier paradigms. Clearly, using anthropomorphic concepts to represent information offers a higher level of data abstraction. On the other hand, variable-based states offer more freedom of expression. Agents interact with each other by communication, that is, by sending and receiving messages. Generally, agents perform or can be seen as performing a *sense–reason–act cycle*: an agent acquires information about its environment (the world and the other agents) by receiving observation results and messages from the other agents, then it reasons to update its beliefs, intentions, plans (or whatever it uses to represent its objectives, information and possible courses of action) and to choose its next actions, and finally it acts, after which the cycle starts over. Information passing in agent-based systems happens through communication. Unlike in the previous paradigms, and in line with the way information is represented in the

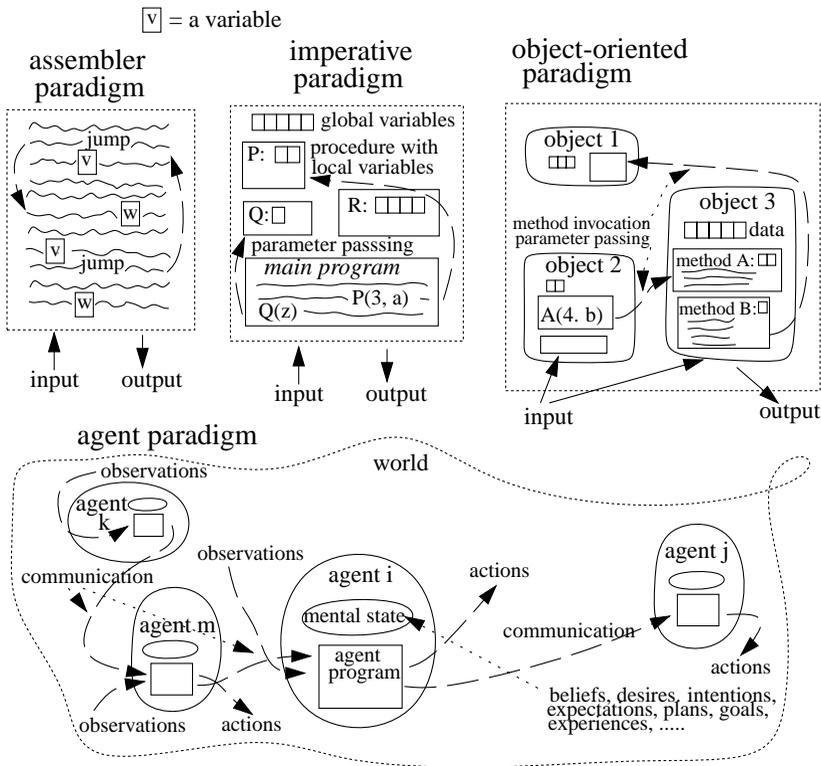


Figure 1.1: The evolution of abstraction in different paradigms

mental states, the information usually is more than a set of variables and values. Messages can be requests for particular information or for assistance in certain activities, or they can contain information (which need not be true) about the environment or the mental state of the sending agent. Expressive agent communication languages have arisen to phrase these messages [46, 133]. Unlike objects, agents don't have to react to messages. As agents are anthropomorphically inspired, they are autonomous, which means that they can sometimes ignore others and just follow their own desires. Agents have total control over their own state and actions. Thus, both data and control are entirely localised.

In Figure 1.1, we summarise the development sketched above in a picture. From assembly to agents, programs and information have become increasingly structured and encapsulated, and information passing has evolved from shared

variables to message passing. For agents, messages to other agents aren't the only means to interact with their environment, as agents also perform actions and observations. The figure shows an evolution from chaotic, fine-grained, unstructured programs to neatly structured, more conceptual programs. Assembly languages provide mnemonic names for statements and variables. Imperative languages offer the possibility to design, name and implement your own data types, and to group together lines of code together into procedures which are named. In object-oriented languages, data and operations on these data are encapsulated in objects, which are also named, such that the methods of a particular object can be called by other objects. And finally, in the agent paradigm, the interaction an agent has with its environment is conceptualised through the notions of observation, action and communication. Internally, the agent is modelled using high-level, anthropomorphic concepts. Thus, information has become more abstract and each piece of information is named to be a belief, desire, intention, or yet something else. Also, the processing of the agent becomes more structured; most agents perform a sense–reason–act cycle, from which they can't deviate.

There is a pattern recognisable in these developments: *structure* is added to order the complex computations, which results in new *concepts*, which are then *named*.<sup>1</sup> This way, new abstractions are created that match the intuitions of a certain paradigm. Each paradigm provides a way of viewing the world, and the abstract concepts used in a paradigm determine the way the programmer will conceive his or her problem specification, the application domain and the program to be built. The successive paradigms increasingly pre-structure the systems built according to these paradigms. As the abstraction level becomes higher, there are more details that the programmer cannot determine anymore. For example, in an agent program, it is not possible to refer to a memory location, while in an assembly program, this is possible (through a variable). As another example, agent programs generally are executed in sense–reason–act cycles, from which the programmer cannot divert, while in object-oriented programs the control of the program can be largely determined by the programmer

---

<sup>1</sup>It seems that this way of approaching reality is quite divinely inspired, as the following Scripture texts about the creation show: "In the beginning God created the heaven and the earth. And the earth was without form, and void; and darkness was upon the face of the deep. . . . And God said, Let there be light: and there was light. And God saw the light, that it was good: and God divided the light from the darkness. And God called the light Day, and the darkness he called Night." (Genesis 1: 1–5) God structures the chaotic earth, creates new concepts (such as light) and names parts of the new reality (day and night). Later on, man is encouraged to do the same: "And out of the ground the Lord God formed every beast of the field, and every fowl of the air; and brought them unto Adam to see what he would call them: and whatsoever Adam called every living creature, that was the name thereof." (Genesis 2: 19)

(except that objects have to respond to messages by executing the method invoked), and programs in assembly allow each possible control flow (through jumps). Though the pre-structuring diminishes flexibility, it provides the system builder with clean-cut, intuitive concepts that aid in constructing compact, powerful software.

In spite of what our historical overview might suggest, the agent paradigm isn't merely the next command-based paradigm. In fact, agent programming also use ideas from other paradigms. The most commonly used other paradigm in this context is *logic programming*. In general, the basic building block of a logic program is a conditional rule, which allows to draw a conclusion when the conditions hold. This style of programming has been used in artificial intelligence for expert systems and knowledge-based systems. As the representations agents use also are logical in nature (there are modal logics that formalise belief, desire and intention, as well as other mental notions the agents might use [21, 34, 85, 102]), a logic programming style fits well with agent programming. In the agent programming language we will define in Chapter 5, we will use a combination of logic programming and imperative programming. The coordination language of Chapter 6 also is inspired by logic (constraint) programming. The work we present in Chapter 2 and 3 assumes that the agent specification language is logical in nature, and in Chapter 4, the local agent state consists of logical formulas. The languages in Chapter 5 and 6 are inspired by *constraint programming*. Here, the values of unknowns are gradually determined by adding constraints on the unknowns. Constraint programming is especially suitable for planning and negotiation, which are important activities for many agents. Another potentially influential paradigm is the *concurrent paradigm*. As agents operate independently, they are concurrent by definition, and so abstractions for synchronisation from concurrent programming can be useful for agents. We will use synchronous communication statements in Chapter 5 and 6, which have been inspired by the concurrent language CSP [61]. The *functional paradigm*, where the basic program unit is function application, might also provide usable concepts for agents, though we haven't seen any agent programming language using ideas from functional programming yet.

The agent paradigm employs a high level of data abstraction and control abstraction. Also, the agent concept doesn't fix a particular style of programming. Thus, starting from anthropomorphic abstractions, the agent paradigm can integrate useful features from different software paradigms.

# Agents

In our view, agents are conceptual entities, designed and/or constructed and/or analysed using anthropomorphic notions, which interact with each other and with the world they are situated in, showing both initiative to achieve their objectives and consideration of the ever changing environment. As is clear from the vagueness of this ‘definition’, we don’t rigidly demarcate the phrase ‘agent’. We don’t demand that agents use specific intentional notions, like belief, desire and intention, in their local (mental) state, nor that agents always are programmed in an agent programming language. In our view, it is perfectly well conceivable that an agent-based system is programmed using an imperative or object-oriented programming language. The most important feature of the agent-oriented paradigm to us is the *agent metaphor*, which allows us to conceive a piece of software as using common-sense, intuitive, anthropomorphic concepts.

Other authors use other definitions of agents. We will give three of these in an attempt to give a sketch of the research field of agents. We start with the view of Jennings, Sycara and Wooldridge, given in [69]:

“For us, then, an agent is a computer system, *situated* in some environment, that is capable of *flexible autonomous* action in order to meet its design objectives.”

This definition is close to our view on the agent concept; to us, situatedness and the ability to flexibly and autonomously interact with the dynamic environment are also essential for agents. The difference between the two agent conceptualisations is that we additionally include the use of anthropomorphic concepts in our definition.

One of the best-known agent definitions is by Wooldridge and Jennings [130]:

“Perhaps the most general way in which the term agent is used is to denote a hardware or (more usually) software-based computer system that enjoys the following properties:

- \* *autonomy* : agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
- \* *social ability* : agents interact with other agents (and possibly humans) via some kind of agent communication language;

- \* *reactivity* : agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;
- \* *pro-activeness* : agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.”

This is the weak notion of agency, according to Wooldridge and Jennings. In the strong notion of agency, an agent is a computer system that

“... in addition to having the properties identified above, is either conceptualised or implemented using concepts that are more usually applied to humans. For example, it is quite common in AI to characterise an agent using *mentalistic* notions, such as knowledge, belief, intention, and obligation.”

This definition emphasises the use of the anthropomorphic agent metaphor, as we also did above.

In the prologue to the textbook on multi-agent systems edited by Weiss, the following agent definition appears:

“An agent is a computational entity such as a software program or a robot that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behaviour at least partially depends on its own experience. As an intelligent entity, an agent operates flexibly and rationally in a variety of environmental circumstances given its perceptual and effectual equipment.” (p. 1, [125])

These definitions, as our own (which we gave at the start of this section), contain many terms which coin intuitive associations but have no definite, well-demarcated meaning. This might seem a weakness of the field. In our opinion, it is one of the attractive features of the agent paradigm, as it clearly shows that multi-agent systems start from intuitions rather than from technicalities. This sets agent-based systems apart from other computer systems. Agents are conceptualised and built starting from a perhaps somewhat hazy vision of an intelligent, interactive, independent and individualised computer process. Even if the agents which presently exist in implemented form aren't yet living up to the high expectations coined by the 'definitions' given above, it is worth while

to have a high-pitched ideal to strive for, which keeps scientists and system builders from getting lost in the symbolic clutter of many computer systems, built using the earlier paradigms.

## Interaction

There is a reason why agent technology seems to be an answer to the question of how to deal with the inherent complexity of modern-day software. To quote the book of Weiss again:

“Modern computing platforms and information environments are distributed, large, open, and heterogeneous. Computers are no longer stand-alone systems, but have become tightly connected both with each other and their users. The increasing complexity of computer and information systems goes together with an increasing complexity of their applications. These often exceed the level of conventional, centralised computing because they require, for instance, the processing of huge amounts of data, or of data that arises at geographically distinct locations. To cope with such applications, computers have to act more as ‘individuals’ or agents, rather than just as ‘parts’. The technologies that distributed artificial intelligence promises to provide are among those that are urgently needed for managing high-level interaction in and intricate applications for modern computing and information processing systems.” (p. 2, [125])

Thus, modern computer application domains often require separate, independent pieces of software to interact with each other and the rest of the environment in order to accomplish some task. The multi-agent metaphor fits this demand like a glove.

In [66], Huhns supports this view. He even coins the phrase ‘interaction-oriented programming’ to describe the type of software engineering which is currently needed.

“Computing is in the middle of a paradigm shift. After decades of progress on representations and algorithms geared toward individual computations, the emphasis is shifting toward *interactions* among computations. The motivation is practical, but there are major theoretical implications. Current techniques are inadequate for applications such as ubiquitous information access, electronic

commerce, and digital libraries, which involve a number of independently designed and operated subsystems. The metaphor of interaction emphasises the autonomy of computations and their ability to interface with each other and their environment. Therefore, it can be a powerful conceptual basis for designing solutions for the above applications.” (p. 29, [66])

According to Huhns, agents are very suitable for interaction-oriented programming, and we agree with this view.

In this thesis, we focus on *interaction aspects* of agents, which means that we don't focus on *internal agent aspects*. This book isn't in the first place about the structure of the mental state of an agent, about the private beliefs, desires and intentions of agents, or about the practical reasoning an agent performs in order to decide what to do next. These aspects have been extensively studied in the agent field, and although there are still many controversies and unsolved issues, we choose to study the inter-agent aspects of the multi-agent paradigm. Agents interact in various ways:

*directly with the environment, through:*

- \* observation
- \* physical action
- \* events (interfering with actions)

*directly with other agents, through:*

- \* communication
- \* coordination
- \* cooperation
- \* group action
- \* distributed planning

*indirectly with each other via the environment, through:*

- \* interfering physical actions
- \* synergy of actions
- \* observing and interpreting behaviour of other agents

In this thesis, we will use the term ‘coordination’ in a broader sense than is usual in the area of multi-agent systems. Coordination in our use means that agents adjust their behaviour to the actions of the other agents and the circumstances and events taking place in the environment, to avoid negative effects or to attain positive effects. This encompasses the more usual meaning of the term, which is avoiding harmful interaction between agents. In our terminology, an agent can coordinate its behaviour with the events taking place in its environment, either to avoid disturbance of its actions by events or to make use of events happening. And agents can coordinate with each other in order to agree on and perform a group action, or to make a distributed plan which they will execute as a cooperating group.

Synergy of actions takes place when the effects of the combination of actions is different from the sum of effects of the separate actions. Synergy is mostly used to designate positive effects. Seen this way, interfering actions and synergetic actions are dual notions.

In this thesis, we study agent interaction. All forms of interaction mentioned in the list above play a major role in one or more chapters of this thesis. We don’t answer one specific question in this book, but explore different features of interaction from different angles of incidence. In order to come to successful interactive agent systems, we need new abstractions, and the work in this book is a contribution to this. To conclude our exploration of interaction in agent systems, we include a quote of Wooldridge and Ciancarini, from their introductory article in the proceedings of a recent workshop on agent-oriented software engineering:

“It is now widely recognised that *interaction* is probably the most important single characteristic of complex software. . . . Many researchers now believe that in the future, computation itself will be understood as chiefly as a process of interaction. This has in turn led to the search for new computational abstractions, models, and tools with which to conceptualise and implement interacting systems.” (p. 1, [129])

## Overview of the thesis

With this thesis, we aim to contribute new abstractions which capture different aspects of agent interaction to the field of agent-oriented software development. As we pointed out earlier, abstraction leads to more powerful and more intuitive concepts. In the agent paradigm the abstract concepts used are inspired

by folk psychology and common sense explanations. We specifically focus on abstract approaches to agent interaction, because interaction is such a central agent feature. By its nature, interaction leads to complexity: when multiple computational processes interact, a combinatorial explosion of execution possibilities results. The new abstract concepts we present in the next chapters aid to make this complexity more manageable.

A danger of using abstraction is that abstract concepts are introduced without grounding them in the computational reality. Just finding intuitive notions is not enough; each notion also has to have a clearly defined meaning. We take care to always relate our abstract notions to lower-level concepts, such that we don't perform abstraction by simply stripping away the underlying dynamics. We structure complex agent interaction and provide intuitive names for the newly identified agent concepts in such a way that the intuitive abstract notions have a formally defined, unambiguous semantics. Thus, we provide foundations for our intuitions about interaction.

The reason we chose the rather broad title 'Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems' for this thesis is that the research accumulated in it explores different issues in the agent field. In the various chapters, we provide formal tools to interpret agent behaviour as motivated by beliefs, desires and intentions (Chapter 2), we perform agent verification (Chapter 3), we construct a generic real-time semantical models of agents (Chapter 4), we design an agent programming language for actions performed by groups of agents (Chapter 5), and we devise a coordination language for distributed planning (Chapter 6). In spite of the diversity of these subjects, *interaction* always is our focus and *abstraction* always is our main tool. In Chapter 4, 5 and 6, we introduce abstract statements to program agent interaction and we relate these statements to the computational reality by defining their formal operational semantics. In Chapter 3, we show how we can abstract from complex, detailed logical formalisations by defining more abstract notions in terms of the lower-level language. And in Chapter 2, we ground the abstract intentional notions of belief, desire and intention in the temporal dynamics of the interaction of the agent with its environment.

We give an overview of the chapters in this book:

**Chapter 1:** *Introduction: What about Abstraction, Agents and Interaction?*

**Chapter 2:** *Abstracting Agent Interaction Histories into Intentionality*

For humans the notions 'belief', 'desire' and 'intention' are intuitive notions to describe behaviour. We use these notions to explain and predict the behaviour of fellow humans, but also of computer applications

(‘Word believes that I want to create an enumerated list here, while in fact I don’t.’). Whenever behaviour gets complex, intentional notions can help to structure and understand it. Because of their intuitiveness and their high abstraction level, intentional notions are particularly useful to describe agent behaviour. They can also be used by agents themselves for this purpose: one agent can ascribe beliefs, desires and intentions, abbreviated as BDI, to another agent, on the basis of observed behaviour, with the purpose to explain and predict its behaviour, and maybe even to manipulate its future behaviour. This chapter addresses the question of how an agent can attribute intentional notions to another agent, on the basis of observed behaviour, to model the other agent’s behaviour.

The intentional stance (as advocated by Dennett; see [25, 26]) states that decisions of agents are based on information about the recent and more distant past; these data can be stored in the form of BDI-formulae and thus determine future behaviour. However, the relationships between such notions and actual observed behaviour of an agent are not always trivial. In this chapter, we lay a connection between the temporal dynamics of the interaction of an agent with its environment (observations received and actions performed over time) and the mental notions belief, desire and intention. These notions thus get a temporal grounding. We define justifying conditions that a (candidate) formula must satisfy in order to qualify as a representation of belief, desire or intention. Using these conditions, intentional notions can be attributed to agents on the basis of externally observed behaviour.

This chapter is an extended version of [76].

### **Chapter 3:** *Reuse and Abstraction in Verification: Agents Interacting with Dynamic Environments*

To make verification a manageable part of the system development process, comprehensibility and reusability of properties and proofs is essential. System developers usually are not extremely skillful in logic, so the properties and proofs have to be as easy and intuitive as possible. We contribute to these objectives by introducing language abstraction, which makes the formulae easier to understand, and with abstract, generic, reusable systems of properties and proofs.

Language abstraction facilitates the communication with the system builders and stakeholders, as verification results in abstracted form can be a readable part of the system documentation. By using two logical languages to phrase properties, an abstract one and a detailed one, the prop-

erties have a well-defined relation to the semantics of the system specification and yet can be intuitively formalised.

Often occurring patterns in agent behaviour can be exploited to establish a library containing properties and proofs. This is illustrated here by verifying action successfulness for the class of single agents acting in dynamic environments. An action is successful if all its expected effects are established. In order to establish action successfulness, agent and environment must interact in a coordinated manner. The properties of the generic system for coordination formalise this proper interaction of an individual agent with its environment. The properties can be used to prove action successfulness for a large class of systems, simply by instantiating the system-specific details into the generic properties.

This chapter is a thoroughly revised and extended version of work published in [74] and [73]. For the running example, we used some material from [72].

#### **Chapter 4:** *An Abstract Model for Agents Interacting in Real-Time*

We offer a new operational model of agents, which focuses on the interaction of agents with each other and with a dynamic environment. In this model, we don't abstract away the dynamic external world, interference of actions and observations that are not accurate due to changes in the world, as our main purpose is to (more) realistically model agent interaction. The model does abstract from the inner workings of agents by offering a definable mental lexicon and a flexible cycle of sensing, reasoning and acting. It consists of a skeleton programming language and its operational semantics; by filling in details of the syntax and semantics, a 'real' agent programming language is obtained. The operational semantics is based on step semantics, which means that each action takes a number of steps, and thus has a duration. The change to the world state during a certain time step is computed by combining the effects of all sub-actions and events taking place then. This way, we can suitably formalise real-time multi-agent behaviour. The model contains a world, which is not totally controlled by the agents and where events take place. Agents can perform individual actions and group actions. Group actions are executed synchronously by the agents participating; this amounts to these agents performing individual actions according to a group action scheme, prescribing the timing of individual actions.

This chapter is an extended and updated combination of the articles [123] and [124].

## **Chapter 5:** *An Abstract Programming Language for Agents Interacting through Group Actions*

Coordination and cooperation are crucial notions in multi-agent systems. Agents can interact with each other in order to discuss and execute coordinated actions to establish objectives which they can't attain individually, or in a less effective way. Proper formalisations for this kind of coordination are scarce in agent programming languages. We provide a constraint programming language called GrAPL, with facilities for group communication, group formation and group collaboration. GrAPL includes three novel statements. Two of these enable groups of agents to communicate about possible constraints on a specific action they might do together. The constraints concern the parameters of the action and the group of agents performing it. If the demands of the agents are compatible, the group reaches an agreement regarding future executions of the action discussed. The third statement is synchronised action execution. Groups of agents can perform an action together, as long as their constraints on the action are satisfied.

The language has a formally defined operational semantics. The three group operations are performed synchronously. GrAPL is (to be) implemented on top of a constraint solver which checks whether the constraints are respected when actions are executed. GrAPL uses ideas from constraint programming to pose constraints on and test the constraints bound to actions. The language largely abstracts from the internal reasoning of agents. We focus on providing new statements of a high abstraction level which facilitate agent interaction (coordination) about and through group actions.

This chapter is an extended version of [122]; in this article, we just presented the syntax of GrAPL, while the chapter also details its operational semantics.

## **Chapter 6:** *An Abstract Coordination Language for Agents Interacting in Distributed Plan-Execute Cycles*

We introduce a coordination language for distributed planning and synchronised plan execution in multi-agent systems. The language enables heterogeneous agents to jointly form a plan by negotiating about constraints on the plan. We abstract from internal agent reasoning, including the planning algorithms the agents employ, and focus on providing high-level coordination statements through which agents can interact to come to a jointly agreed plan, which they subsequently can execute. The

statements of our language enable agents to form groups dedicated to constructing a distributed plan together, to communicate about details of the plan (as produced by the planning algorithms used by the agents), to jointly decide that the plan is sufficiently elaborated by committing to it, and finally to execute it.

The coordination language has a formal operational semantics. The language is constraint-based, like the programming language GrAPL. The constraints pertain to the objective of the plan and the circumstances in which the plan will be executed, the group of agents negotiating about the plan, the actions that are part of the plan, the actors that will execute the actions and the order of the actions. Agents thus negotiate about and execute structured plans instead of simple group actions. The statements of the coordination language enable system builders to program coordination between agents doing distributed planning in a neat and abstract manner.

Moreover, we propose a coordination architecture for multi-agent systems. This architecture provides support for managing the constraints on plans by means of a constraint solver and provides control mechanisms for the distributed execution of plans.

The work in this chapter has not been published previously.

**Chapter 7:** *Conclusion: Abstract Interaction is what Agents are for!*

## CHAPTER 2

---

# Abstracting Agent Interaction Histories into Intentionality

---

*The only baggage you can bring  
Is all that you can't leave behind*  
U2

### 2.1 Introduction

As agents become ever more sophisticated and their behaviour often goes beyond purely reactive behaviour, means are needed to understandably describe and predict their actions. A general human tactic to understand the behaviour of their fellow human beings is to ascribe mental notions to them. For example, when you see a man running along a train platform, you will probably think that this man *wants* to catch a train of which he *believes* that it's almost departing. This conclusion abstracts the interaction of the man with his environment into simple, familiar concepts. An attractive feature of the agent paradigm is that it often uses anthropomorphic, intentional notions like those

in the example to describe system behaviour. These notions offer a high level of abstraction and have intuitive connotations, because of their folk-psychological nature. While there is no generally accepted definition of agent systems, several authors [21, 85, 102] hold *informational* and *motivational* attitudes as useful attributes of an agent. Informational attitudes usually are the notions of belief and knowledge, and motivational attitudes are concepts like desire, intention, goal, plan, obligation and choice. There is no fixed set of intentional concepts; while the main ideas are similar, different approaches make different choices. Nevertheless, belief, desire and intention, abbreviated as BDI, seem to constitute a classical kernel set of mental notions.

In [25, 26], Dennett puts forward the *intentional stance* as opposed to explanations from a direct physical perspective (the physical stance). According to the intentional stance, an agent is assumed to decide to act and communicate based on its beliefs about its environment and its desires and intentions. These decisions, and the intentional notions by which they can be explained and predicted, generally depend on information acquired by observations and communication in the past and not only on information just acquired. The role of the intentional notions often is to extract and maintain relevant features of the agent's history. To be able to analyse the occurrence of intentional notions in the behaviour of an observed agent, the observable behavioural patterns over time form an empirical basis. We present means to abstract these patterns in the interaction history of the agent with its environment into intentional notions. According to Dennett, the intentional stance provides folk-psychological means to compactly and intuitively describe system behaviour, at a level that sufficiently abstracts from physical details that clutter understanding.

Dennett's account is philosophical and informal in nature. In this chapter, we formalise Dennett's ideas by providing justifying conditions for intentional notions. In the formalisation introduced below, the temporal aspect of the dynamics of the interaction with the environment is made explicit and related to mental notions. We propose justifying conditions for the notions of belief, intention and desire. The formal criteria describe the relation between the intentional notions and the externally observable behaviour of an agent. If we attribute belief, desire or intention to somebody else, then we do this based on the actions we see this person doing, and also on the information we presume that this person must have obtained by observation and communication. For example, suppose we are in the main post office in Utrecht. If you enter this post office, you have to take a number to be served. We enter, just following a woman in a yellow dress. She has taken a number, which we observe to be 358. We both sit down somewhere to wait for our turn, watching the electronic board indicating whose turn it is. We see that this board currently shows num-

ber 214. We then assume the woman in the yellow dress also saw this. If, after five minutes, we see the woman impatiently leaving the post office, while the board indicates 218, then we can ascribe to her the desire to be served quickly or not at all. Similarly, the justifying conditions we introduce relate intentional notions to observed and communicated information the agent receives and to actions the agent decides to perform.

Received information (observed or communicated), and decisions to perform specific actions (including communicative and observative actions), constitute the input and output interfaces of an agent to the environment in which the agent functions. Externally observed behaviour traces of the agent are formalised as temporal sequences of the agent's input and output states. We use a temporal trace language to express properties of behaviour. In this language, a temporal formula in terms of the agent's input and output states defines a class of (possible) histories of interaction with the environment. The formal criteria express when such a formula defines a class of interaction histories that can be related to a specific belief, desire or intention. A temporal formula satisfying these criteria for a specific intentional notion is called an *external representation* or *temporal grounding* of this notion. If a formula, describing a specific behaviour pattern of an agent, is an external representation of an intentional notion, and the formula holds for a certain agent behaviour, then we can *ascribe* the intentional notion to that agent. Note that the formal criteria are useful both for agents that have explicit internal representations of BDI and for agents that don't have these. Using the justifying conditions, internal representations can be grounded in the agent's behaviour and intentional notions can be attributed to agents without internal representations of belief, desire and intention by definition in terms of external behaviour.

The formal analysis of intentional notions in agents as performed in this chapter can be compared to the cognitive approach in the study of animal behaviour. A biologist trying to interpret animal behaviour can first gather a large amount of data on the actions of the animal in various situations by field studies and experiments. Based on this empirical data, he can try to find an explanation of why the animal acts like it does, and then go on testing this explanation in new situations. This explanation is given in terms of internal mental representations. The cognitive approach has been severely criticised by behaviouristic biologists, as they disapprove of presupposing cognitive representations in animals. The approach followed in this chapter has similarities with the cognitive approach: based on a set of externally observed behaviour traces of an agent, high-level notions such as beliefs, desires and intentions are attributed in such a way that an easy to understand explanation can be given of observed behaviour, and behaviour can be predicted. But our approach differs from the cognitive

approach, in that we *don't presuppose* mental representations in agents, and that we have a *detailed formalisation* of the attribution process, which lacks in the cognitive approach. Also, we provide criteria that fix the connection between the attributed intentional notions and the behaviour of the agent, making the attribution of notions a testable and rigorous process.

Ascription of beliefs, desire and intention can be done in various contexts. In this chapter, we focus at organisation modelling. In organisations, behaviour is assumed to be constrained by the organisational structure (cf. [43, 44]), including, in particular, behavioural role specifications (cf. [45]). These role specifications prescribe to a certain extent the dynamics of the organisation, in order to have well-coordinated processes in the organisation. A role specification usually does not completely fix behaviours, but often allows some free space for personal initiative. This freedom also may provide possibilities for agents to avoid certain behaviours as expected by others, negatively effecting the degree of coordination in the organisation. If avoidance behaviour is possible in an organisational structure, it is useful when agents fulfilling a certain role in the organisation can reason about the possible behaviour of the agents in other roles, for example using the intentional stance. For example, to an agent functioning within an organisation it may be helpful to have capabilities to predict in which circumstances certain inappropriate desires or intentions are likely to arise as a basis for the behaviour of a colleague within the organisation, either to avoid the arising of these intentions by preventing the occurrence of circumstances that are likely to lead to them, or if these circumstances cannot be avoided, by anticipating consequences of the intentions. Similarly, for cases that appropriate desires or intentions may or may not arise depending on circumstances, interpretation and prediction of the behaviour of colleagues makes sense. Such capabilities of *anticipatory reasoning* about the behaviour of colleagues in an organisation are quite important for an organisation to function smoothly. More specific examples can be found in Section 2.6 below. This chapter gives a formal basis for these types of anticipatory reasoning.

In Section 2.2 we discuss background material from various research areas. Section 2.3 informally discusses the assumptions made on the notions belief, desire and intention, and the way they interact with each other and with external notions. In Section 2.4 the formal temporal trace language used in this chapter is introduced. Section 2.5 is the formal heart of the chapter; here, criteria formalising the relation between the intentional notions and the external behaviour of an agent are defined. In Section 2.6 an example application to organisation modelling is addressed. Section 2.7 is a discussion.

## 2.2 Background

Intentionality is presented from several perspectives in agent related literature. In this section, we will illustrate the positions taken by several influential authors from philosophy, computer science and artificial intelligence. Also, I review a less famous author by relating my master thesis to the present issues in this chapter.

In their famous overview article [130], Wooldridge and Jennings advocate the usage of animistic, intentional explanations of the behaviour of agents. They even hold intentionality a necessary condition for agent-hood, as this quote shows:

An agent is a system that is most conveniently described by the intentional stance; one whose simplest consistent description *requires* the intentional stance.

So, a system which can be easily explained using other means than intentional concepts is not an agent system, according to Wooldridge and Jennings. We tentatively agree with this opinion. Agents generally don't just react to impulses from their environment, but also take initiatives to achieve conditions in their environment. That is, agents show pro-active as well as reactive behaviour. In order for an agent to have non-reactive (that is pro-active, initiative-taking) behaviour, its internal reasoning process must use other motivational attitudes (for example goals and/or intentions) besides beliefs to decide on the actions to be done. These other motivations are not based primarily on the information the agent has about the environment, but on the aims the agent wants to realise in its environment.

The level of abstraction needed to give an intuitive explanation of a mechanism or system is indicative for the level of complexity of the system. To quote Wooldridge and Jennings again:

Put crudely, the more we know about a system, the less we need to rely on animistic, intentional explanations of its behaviour. However, with very complex systems, even if a complete, accurate picture of the system's architecture and working *is* available, a mechanistic, *design stance* explanation of its behaviour may not be *practicable*. Consider a computer. Although we might have a complete technical description of a computer available, it is hardly practicable to appeal to such a description when explaining why a menu appears when we click a mouse on an icon. In such situations, it may be more appropriate to adopt an intentional stance

description, if that description is consistent, and simpler than the alternatives. The intentional notions are thus *abstraction tools*, which provide us with a convenient and familiar way of describing, explaining, and predicting the behaviour of complex systems.

The ascription of mental attitudes to computer programs such as agents has been criticised as being mere anthropomorphism. By pretending computer programs have goals, plans and desires, just like humans, they seem more intelligent, while in fact the functionality isn't any higher. On the one hand, this criticism is justified; just by giving things a different name, software isn't getting any smarter. But when intentionality is used as an abstraction tool to describe reasoning behaviour of a certain level of complexity and ingenuity, this criticism isn't justified. Using anthropomorphic, intentional notions is a good way to make agent behaviour explainable. The more complex the software, the more useful intentional notions can be. John McCarthy supports this view, as the following quotation<sup>1</sup> shows.

To ascribe beliefs, free will, intentions, consciousness, abilities, or wants to a machine is legitimate when such an ascription expresses the same information about the machine that it expresses about a person. It is *useful* when the ascription helps us understand the structure of the machine, *its past or future behaviour*, or how to repair or improve it. It is perhaps never *logically required* even for humans, but expressing *reasonably briefly* what is actually known about the state of the machine in a particular situation may require mental qualities or qualities isomorphic to them. Theories of belief, knowledge and wanting can be constructed for machines in a simpler setting than for humans, and later applied to humans. Ascription of mental qualities is *most straightforward* for machines of known structure such as thermostats and computer operating systems, but is *most useful* when applied to entities whose structure is incompletely known. (McCarthy, in [89])

Note that McCarthy thinks the ascription of intentional qualities can aid in understanding past or future behaviour of a machine (or an agent). This is one of our starting points, as we explained in the introduction.

Dennett also advocated the use of higher abstraction levels to explain system behaviour. In [25, 26], Dennett introduces the *design stance* and the *intentional stance* as two viewpoints that abstract from the details of the

---

<sup>1</sup>This fragment is quite popular. We found it in [130], where Wooldridge and Jennings repeat McCarthy's words as Shoham quoted them in his article on agent-oriented programming [113].

*physical stance*. Different description levels with ontologies for emerging patterns in the simulation environment Life are used to explain the advantage of higher levels; cf. [25], pp. 37-39; [26], pp. 37-42. In addition, Dennett uses the description levels in computer systems (actually of a chess computer), embedded (and hence visualised) in the two-dimensional Life environment as a metaphor to explain the advantage of design stance and intentional stance explanations for mental phenomena over physical stance explanations:

The scale of compression when one adopts the intentional stance toward the two-dimensional chess-playing computer galaxy is stupendous: it is the difference between figuring out in your head what white's most likely (best) move is versus calculating the state of a few trillion pixels through a few hundred thousand generations. But the scale of savings is really no greater in the Life world than in our own. Predicting that someone will duck if you throw a brick at him is easy from the folk-psychological stance; it is and will always be intractable if you have to trace the protons from brick to eyeball, the neurotransmitters from optic nerve to motor nerve, and so forth. [26], p. 42

The intentional notions in this philosophical account aren't formalised. *Agent theories* [21, 85, 102] provide different logical formalisations of several intentional notions. Each of the logics, which are reviewed in [121], focuses on a different set of informational and motivational attitudes, formalises these in modalities, and provides properties of the notions by axiomatising them. These theories are also known as BDI-logics. The logics generally provide logical relations between the different intentional notions, focusing more on static aspects than on the dynamics of the intentional notions. The logical theories mostly don't relate the history of an agent's past actions and observations to its present internal motivations.

The temporal dependencies between the different intentional notions and between the intentional notions and the observable behavioural 'real world' patterns, are thus only covered partially in the literature on BDI-logics as mentioned: within a BDI-logic, for a given world state all beliefs, desires and intentions are derived at once, without internal dynamics. In other references from the area of cognitive science and philosophy of mind, this omission has been criticised, and instead a different perspective is proposed, where dynamics of mental states and their interaction with the environment are central; e.g. [9, 19, 20]. For example, in [9] Bickhard emphasises the relation between the (mental) state of a system (or agent) and its past and future in the interaction with its environment:

When interaction is completed, the system will end in some one of its internal states — some of its possible final states. Some environments will leave the system in that same final state, when interactions with this system are complete, and some environments will leave the system in different possible final states. The final state that the systems ends up in, then, serves to implicitly categorise together that class of environments that would yield that final state if interacted with. A possible final state, then, implicitly defines, in an interactive sense, its class of environments. Dually, the set of possible final states serves to differentiate the class of possible environments into those categories that are implicitly defined by the particular final states. The overall system, with its possible final states, therefore, functions as a *differentiator* of environments, with the final states implicitly defining the differentiation categories. (...) Representational content is constituted as indications of potential further interactions. (...) The claim is that such differentiated functional indications in the context of a goal-directed system constitute representation — emergent representation.

This suggests that mental states need to be grounded in interaction histories on the one hand, and have to be related to future interactions on the other hand. However, in this literature no formalisation is proposed based on this perspective. In the formalisation introduced below, the temporal aspect of the dynamics of the interaction with the environment is made explicit and related to mental notions. In [71], Jonker and Treur propose a formalisation of Bickhard's philosophical interactivist perspective.

Returning to the area of agent research, we briefly mention agent programming languages and -architectures. There are several agent programming languages and architectures based on the mental perspective, where agents have beliefs, goals and plans. Examples of programming languages are [59, 101, 113]. Agents written in these languages execute a sense–reason–act-cycle; they *sense* their environment through observation and communication and update their beliefs, they *reason* to keep their plans and goals up-to-date and to decide which actions to perform next, and then they *execute* the chosen *actions*, after which the cycle starts again. Agent architectures with intentional inspiration are for example IRMA [13] and GRATE\* [67]. As agent programming languages and agent architectures are meant to actually build agents that interact with their environment, mental notions in programming languages and architectures *are* related to external agent actions and observations, in contrast

to agent theories, where this is usually neglected. Unfortunately, the relation between internal intentional notions and external agent behaviour is often not formally described.

The final background perspective we review is taken from my master thesis, called “Motivated Machines and Longing Logics” [121]. In this report, I review the work of the philosopher Nelson [94] in particular. Nelson is a mechanist, which means that he believes the human mind functions according to a system of rules and is thus nothing but a machine. Nelson thinks the mind essentially is the same as a nondeterministic finite automaton. To give evidence for this thesis, Nelson then tries to define finite automata that implement *belief* and *desire*. Though I show in [121] that there are many flaws and weaknesses in Nelson’s account, his conception of desire is very interesting. According to Nelson, an agent *desires* something if, whenever it *thinks* there is an *opportunity* to perform an action to get what it wants, it *acts*, and the result of its act is that it *expects* to have what it desired. Note that this notion of desire is quite strong, as each desire leads to actions if there are opportunities for these actions. Other authors generally take desire to be weaker; a desire only leads to actions if an agent chooses to make it into a goal and adopts intentions to perform actions for achieving this goal. Nelson’s desires are as strong as goals and/or intentions are in other work. Also, it is unusual to include expectation in intentional definitions. As will become clear in the next sections, we use a definition for intention that is very close to Nelson’s desire definition, minus the expectation part.

## 2.3 Basic assumptions

In this section, we informally discuss the notions of belief, desire, and intention, and their interdependencies, as shown in Figure 2.1. The solid arrows in this figure show how mental notions depend on other mental notions; for example, an intention to do some action arises from a desire for something and the belief that there is a reason to pursue this desire by doing the action. The dashed arrows indicate that *part* of the history of an agent can lead to desires or beliefs; this doesn’t involve every aspect of the history. The interdependencies in Figure 2.1 are based upon a number of assumptions on beliefs, desires, and intentions. These assumptions have been inspired by the work of Dretske on the understanding of behaviour [31]. The assumptions we make keep the notions relatively simple; the approach can be extended for more complex notions.

*Assumptions on beliefs.* In the simplest approach, beliefs ( $\beta$  in Figure 2.1) are just based on information the agent has received by observation or commu-

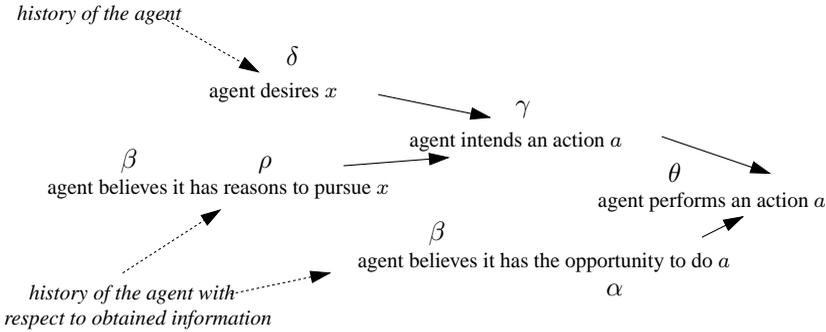


Figure 2.1: Relationships between the BDI notions

nication in the past, which has not been overridden by more recent information. This entails the first assumption: if the agent has been informed in the past about a world fact, and no opposite information has been received since then, then the agent believes the world fact. The second assumption is the converse: for every belief of a world fact, there was a time at which the agent was informed about this world fact (by sensing or communication), and no opposite information was received since then. These assumptions can also be replaced by less simple ones, if required. For example, in some domains, observation is not reliable and some of the other agents provide untrue information, and so the agent can only believe information if a reliable agent communicates it, or if the same information is provided from two different sources. Note that we restrict ourselves for the sake of simplicity to *perceptual* beliefs in this chapter; we don't consider beliefs the agent obtains by reasoning. For this kind of belief other assumptions are needed.

*Assumptions on intentions.* In the first place, when an agent performs ( $\theta$  in Figure 2.1) a physical or communicative action ( $a$  in the figure), we assume the agent has intended ( $\gamma$ ) to do that. The agent thus doesn't perform actions by accident. Secondly, we assume that an agent who intends to perform an action will execute the action when it believes that an opportunity ( $\alpha$ ) occurs. The agent can initiate the action some time after it believes there is an opportunity, because its reasoning processes could take some time. Note that these two assumptions closely match Nelson's definition of desire, as discussed in the previous section.

*Assumptions on desires.* The first assumption on desire is that every intention is based on a desire ( $\delta$ ). An agent can desire states of the world as well as

actions to be performed; so,  $x$  in the figure can be a world state or an action. If the desire depicted in Figure 2.1 is for an action, then  $x$  and  $a$  coincide ( $x = a$ ). When the agent has a set of desires, it can choose to pursue some of them. A chosen desire for a state of the world can lead to an intention to do an action if, for example, expected effects of the action (partly) fulfil the desire. The second assumption is that, given a desire, for each relevant action there is an additional reason ( $\rho$ ), so that if both the desire is present and the agent believes the reason, then the agent will intend to perform the action. In case the desire is for doing an action, then  $\rho$  merely selects desires which are to be pursued; in case the desire is for achieving a world state, the reason  $\rho$  both selects a desire and picks an action that can lead to its fulfilment.

*Determinism Assumption.* Another basic assumption we make is that a software agent's internal states functionally depend on the history of the agent; i.e., two copies of the same agent build up exactly the same (internal) states if they have exactly the same histories of input. For a software agent, running on a deterministic machine, the Determinism Assumption can be considered a reasonable assumption. Differences between the behaviours of two copies of the same software agent will be created by their different histories. For most of the concepts defined below, this assumption is not strictly necessary. However, it is an assumption that strongly motivates the approach. If determinism is assumed it makes sense to exploit temporal formulae that describe the history of the agent as candidates for representations of externally attributed intentional notions; otherwise these formulae will not be found.

## 2.4 Formal preliminaries

To be able to formulate properties of agent behaviour, we need a formal temporal language. In this section, we define and explain this language. We base the language in representations used in agent systems. No matter what architecture or programming language is used, an agent needs data structures for observation results and incoming communication and for actions the agent has decided to do. Different agent systems use different representations. In this chapter, we assume the agent is logic-based, and that all information the agent handles is logically formulated. For example, if an agent decides to do action  $a$ , this is represented by the atom `to_be_performed(a)` getting the truth value *true*.

The agents we consider have three different information spaces, as Figure 2.2 shows. The *input interface* contains information coming in from the environment, that is, the external world and the other agents. The *output interface* contains representations of actions the agent is about to do; these actions can

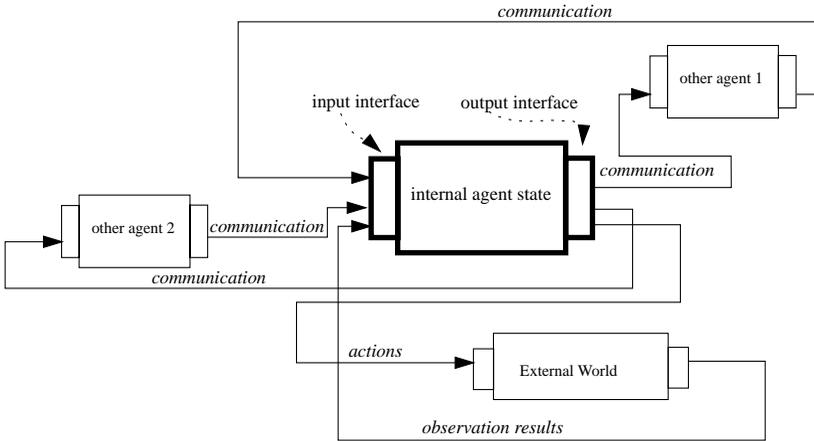


Figure 2.2: An agent in its environment

be physical, communicative or observative. When the agent executes a physical action, the atom representing the action is transferred to the external world, where the actual execution takes place. Observations are processed similarly; the atom representing the observative action is transferred to the external world, which then returns the observation result to the agent. Note that performing a communicative action is the same as sending a message, as is the case in this agent model. The third information space is the *internal agent state*. If the agent internally employs BDI-notions, then beliefs, desires and intentions are stored in this space. The internal agent state can contain whatever information the agent programmer holds useful for the reasoning processes of the agent. For example, the agent could maintain a history of observations performed in the past, or it could have expectations about the outcomes of actions it is doing. For our present purposes, these choices are not relevant.

In the specification of agent programs formal ontologies (i.e., vocabularies) for the agent's input, output and internal state are used and formulae based on these ontologies. We adopt these ontologies as the basis of the temporal language for describing agent behaviour. For simplicity, we use predicate logic to specify both ontologies and formulae. An ontology is specified as a finite set of sorts, constants within these sorts, and relations and functions over these sorts (sometimes also called a signature). By applying the relations and functions to the constants, ground atoms are obtained, which form the basic constituents from which formulae can be constructed. The union of two ontologies contains

the sorts, relations and functions from both ontologies and is again an ontology. If functions are used, recursion is excluded, to keep the number of ground atoms finite. By considering the finite set of ground atoms as proposition symbols, the logical languages based on it can be treated as propositional.

### 2.4.1 State Language

First, we use an ontology for facts concerning the actual state of the external world: ontology EWOnt. For example, in a system modelling an artificial mouse looking for food in its environment, EWOnt contains constants like mouse and cheese, and relations like `at_position(..., ...)`. A formula based on this ontology is for example `at_position(mouse, p_2) ∧ at_position(cheese, p_1)`. Some of the other (agent) ontologies will make use of EWOnt, as we will show later on. Next, we have ontologies for the three agent information spaces. These ontologies form the basis of a language to describe the state of the agent. The *agent input ontology* InOnt contains constructs for observation results and communication received. The following *input atoms* are used. The observation result that  $\varphi$  holds is denoted by `observation_result( $\varphi$ )`, where  $\varphi$  is describing information on the external environment. This means that  $\varphi$  is a formula based on the ontology EWOnt. Similarly, `communicated_by( $\varphi$ , i)` denotes that agent  $i$  has communicated  $\varphi$ . The *agent output ontology* OutOnt contains constructs to represent decisions to do actions within the external world, as well as constructs for outgoing communication and observations that the agent needs to obtain. The following *output atoms* are used: `to_be_performed(a)` denotes that the agent decides to perform action  $a$ , `to_be_communicated_to( $\varphi$ , i)` means that the agent sends information  $\varphi$  to agent  $i$ , and `to_be_observed( $\varphi$ )` denotes that the agent decides to perform an observation to investigate the truth of  $\varphi$ . All expressions introduced to formalise the interaction of the agent with its environment are meta-expressions; some of their arguments refer to statements in an object-level language. In the above expressions, the symbol  $\varphi$  refers to a formula based on EWOnt. So,  $\varphi$  could be `at_position(cheese, p_1)`, and then `to_be_observed(at_position(cheese, p_1))` denotes that the agent wants to perform an observation regarding the position of the cheese. The *internal agent ontology* InternalOnt is used for the internal (e.g., BDI) notions. The *agent interface ontology* is defined by `InterfaceOnt = InOnt ∪ OutOnt`; the *agent ontology* by `AgOnt = InterfaceOnt ∪ InternalOnt`, and the *overall ontology* by `OvOnt = AgOnt ∪ EWOnt`.

The logical language based on the ontologies defined above is called the *state language* (abbreviated as SL) and its formulae are called *state formulae*. State formulae describe the state of an agent or of the world at one partic-

ular (unmentioned) point in time. Existential and universal quantification is allowed in formulae. So, the formula  $\exists p \in \text{POSITION} : \text{at\_position}(\text{cheese}, p)$  is a formula of SL, and means that there is cheese at some position. As there are different ontologies, there are different state languages, each based on an ontology. All state formulae based on a certain ontology  $\text{Ont}$  constitute the language  $\text{SL}(\text{Ont})$ . SL is the overall state language; it is an abbreviation of  $\text{SL}(\text{OvOnt})$ . Note that  $\text{SL}(\text{Ont}) \subseteq \text{SL}$  for each ontology  $\text{Ont}$ . An *information state* of the input of the agent is an assignment of truth values  $\{\text{true}, \text{false}, \text{unknown}\}$  to the set of ground atoms in  $\text{SL}(\text{InOnt})$ . The set of all possible information states of the agent’s input is denoted by  $\text{IS}(\text{InOnt})$ . Similarly,  $\text{IS}(\text{OutOnt})$  and  $\text{IS}(\text{InterfaceOnt})$  are the sets of all possible information states of the agent’s output state and internal state. The set of information states of the overall agent then is  $\text{IS}(\text{AgOnt})$ , which we usually abbreviate to  $\text{IS}(\text{Ag})$ . We define the set of information states of the external world  $\text{IS}(\text{EWOnt})$  (abbreviated to  $\text{IS}(\text{EW})$ ) and of the overall system  $\text{IS}(\text{OvOnt})$  (abbreviated to  $\text{IS}(\text{S})$ , where S refers to system) analogously.

## 2.4.2 Temporal Language

To describe behaviour of agents, we refer to time in a formal manner. For our approach, it isn’t relevant whether time is dense or discrete. The only restriction we need is the *finite variability assumption*, which we provide in the next subsection. For now, we assume the time frame is the set of natural numbers or a finite initial segment of the natural numbers. The finite variability assumption basically means the time frame is isomorphic to one of these. An *overall trace*  $\mathcal{M}$  over a time frame  $\mathbb{T}$  is a sequence of information states  $(\mathcal{M}^t)_{t \in \mathbb{T}}$ , where  $\mathcal{M}^t \in \text{IS}(\text{S})$ . We also have traces of an agent, or of the external world; then, the information states pertain to the agent or the world instead of the whole system. A *temporal domain description*  $\mathcal{W}$  is a set of overall traces. The information on agent behaviour in the temporal domain descriptions can be compared to the information a biologist gathers on an animal by repeatedly studying its behaviour in various circumstances. There can be many temporal domain descriptions for an agent system, as there are many sets of system traces possible. In order to give a well-founded explanation of the agent behaviour, it is necessary to carefully select a world description which is large enough and diverse enough to be a good representation of all possible agent behaviours in each possible situation. Given a trace  $\mathcal{M}$  of agent Ag, the information state of the input interface at time point  $t$  is denoted by  $\text{state}(\mathcal{M}, t, \text{input}(\text{Ag}))$ . Analogously,  $\text{state}(\mathcal{M}, t, \text{output}(\text{Ag}))$  denotes the information state of the output interface of the agent at time point  $t$ , and  $\text{state}(\mathcal{M}, t, \text{internal}(\text{Ag}))$  the internal

information state. We can also refer to the overall information state of a system (agents and environment) at a certain moment; this is denoted by  $\text{state}(\mathcal{M}, t)$ . These formalised information states can be related to state formulae via the formally defined satisfaction relation  $\models$ . If  $\varphi \in \text{SL}(\text{InOnt})$ , then

$$\text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models \varphi$$

denotes that  $\varphi$  is true in this state at time point  $t$ , based on the strong Kleene semantics (e.g., [10]). The sorted predicate logic temporal language TL is built on atoms like the one above, using the usual logical connectives and quantification (for example, over traces, time and state formulae). This structure of TL is comparable to the approach in the situation calculus, where formulas are built from atoms like  $\text{holds}(\varphi, \text{S})$ , which informally translates to “In situation S, formula  $\varphi$  is true”. TL is the language we use to formalise properties of the dynamic behaviour of agents. An example of a formula of TL is

$$\begin{aligned} \forall t_1 : \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models & \\ & \text{to\_be\_performed}(\text{kick\_ball\_through\_neighbor\_window}) \\ \Rightarrow & \\ \exists t_2 > t_1 : \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models & \\ & \text{observation\_result}(\text{angry\_neighbor\_at\_door}) \end{aligned}$$

The meaning of this property is straightforward. In contrast with SL, which is a three-valued logic, TL is a two-valued logic; an atom can either be true ( $\text{state}(\mathcal{M}, t) \models \varphi$ ) or false ( $\text{state}(\mathcal{M}, t) \not\models \varphi$ ). The reason that this is so is the way we defined information states; an information state is total, in the sense that it fixes a truth value (*true*, *false* or *unknown*) for each ground atom of the state language involved. Thus, the truth value of a composite state formula  $\varphi \in \text{SL}$  in an information state  $\text{state}(\mathcal{M}, t)$  can always be computed to be either *true*, *false* or *unknown*. In case the truth value is *true*, the temporal atom  $\text{state}(\mathcal{M}, t) \models \varphi$  has truth value *true*; if the truth value of  $\varphi$  in the state is *unknown* or *false*, then  $\text{state}(\mathcal{M}, t) \models \varphi$  has truth value *false*. These are the only two options. In other words, there is no way to construct a temporal atom  $\text{state}(\mathcal{M}, t) \models \varphi$  which has truth value *unknown*. Note that if  $\varphi$  is *unknown* in the state, then both  $\text{state}(\mathcal{M}, t) \models \varphi$  and  $\text{state}(\mathcal{M}, t) \models \neg\varphi$  have truth value *false*. If a composite temporal formula  $\psi \in \text{TL}$  is constructed from temporal atoms like these, the truth value can be determined using traditional semantics for two-valued logics.

Sometimes, we employ the notation  $\psi(\mathcal{M}, t)$  to refer to a formula from TL in which trace  $\mathcal{M}$  and time point  $t$  occur. We allow additional language elements as abbreviations of formulae of the temporal language.

To focus on different aspects of the agent and time, we need ways to restrict traces. Restrictions have two parameters, one for the ontologies and one for the time interval. The ontology parameter indicates which parts of the agent are considered. For example, when this parameter is  $\text{InOnt}$ , then only input information is present in the restriction. The time interval parameter specifies the time frame of interest. To illustrate this, the notation  $\mathcal{M}_{[0,t]}^{\text{OutOnt}}$  denotes the restriction of  $\mathcal{M}$  to the past up to  $t$  and to output atoms. The *restriction*  $\mathcal{M}_{\text{Interval}}^{\text{Ont}}$  of a trace  $\mathcal{M}$  to time in *Interval* and information based on  $\text{Ont}$  is defined as follows:

$$\mathcal{M}_{\text{Interval}}^{\text{Ont}}(t)(\alpha) = \begin{array}{ll} \mathcal{M}(t)(\alpha) & \text{if } t \in \text{Interval} \text{ and } \alpha \text{ is a ground atom} \\ & \text{of Ont} \\ \text{unknown} & \text{otherwise} \end{array}$$

In the next section, we will give criteria that relate beliefs, desires and intentions at a certain point in time to formulae describing the history of the agent up and until that point in time. Therefore, it is useful to define the category of past formulae. A *past formula* for  $\mathcal{M}$  and  $t$  is a temporal formula  $\psi(\mathcal{M}, t) \in \text{TL}$  such that each time variable different from  $t$  is restricted to the time interval before  $t$ ; no reference is made to time points beyond  $t$  in  $\psi(\mathcal{M}, t)$ . In other words, every time quantifier for a variable  $t'$  is bounded by a restriction  $t' \leq t$ , or  $t' < t$ , or is within the scope of another variable  $t''$  and bounded by a restriction  $t' \leq t''$  or  $t' < t''$ . This way, no variable in  $\psi(\mathcal{M}, t)$  refers to a time later than  $t$ . The set of past formulae over ontology  $\text{Ont}$  with respect to  $t$  is denoted by  $\text{PL}(\text{Ont}, t)$ . Note that for any past formula  $\psi(\mathcal{M}, t)$  it holds:

$$\forall \mathcal{M} \in \mathcal{W} \forall t \quad \psi(\mathcal{M}_{[0,t]}, t) \Leftrightarrow \psi(\mathcal{M}, t)$$

This is the case because  $\psi(\mathcal{M}, t)$  describes a property of the part of trace  $\mathcal{M}$  before  $t$ ; whether this property holds thus only depends on the time interval  $[0, t]$ .

### 2.4.3 Finite variability and change pinpoint principle

In the previous subsection, we mentioned that the time frame  $T$  can be either discrete or dense. In case the time frame is dense, we need some provisions to ensure that traces can be sliced up into intervals such that the overall information state is stable within each interval. A property that guarantees this is *finite variability* (we use a variant of the one in [5]). This property forbids gradual changes in information states, like for example the changes to the position of a ball falling down. Traces with a dense time frame that adhere to finite variability are isomorphic to traces where time is discrete, because the moments that

states change can be related to discrete time points. To be precise, the finite variability assumption defines a specific subset of all linear time traces, namely only those traces where between any two time points only a finite number of changes occur, and after each change, a first time point exists for the new state:

$$\begin{aligned} \forall s, t \forall \varphi : s < t \quad &\Rightarrow \exists n \in \mathbb{N} \setminus \{0\} \exists t_1, \dots, t_n : s = t_1 < \dots < t_n = t \wedge \\ &\forall i \in \{1, 2, \dots, n-1\} \forall t' \in [t_i, t_{i+1}) : \\ &[(\text{state}(\mathcal{M}, t_i) \models \varphi \Leftrightarrow \text{state}(\mathcal{M}, t') \models \varphi) \wedge \\ &(\text{state}(\mathcal{M}, t_i) \models \neg\varphi \Leftrightarrow \text{state}(\mathcal{M}, t') \models \neg\varphi)] \end{aligned}$$

Traces satisfying this adhere to the following *change pinpoint principle*, as we will prove in the continuation of this subsection. The change pinpoint principle expresses that any change in the state of the agent system can be related to a unique time point: for each change that occurs in a trace  $\mathcal{M}$  it should be possible to point at a time point *where this change has just occurred*:

$$\begin{aligned} \forall \varphi \forall t_3 : \quad &\text{state}(\mathcal{M}, t_3) \models \varphi \Rightarrow \\ &\exists t_2 \leq t_3 \forall t \in [t_2, t_3] : \text{state}(\mathcal{M}, t) \models \varphi \wedge \\ &(t_2 \neq 0 \Rightarrow \exists t_1 < t_2 \forall t' \in [t_1, t_2) : \text{state}(\mathcal{M}, t') \not\models \varphi) \end{aligned}$$

This means that it is always possible to find a moment in time where a formula has just become true. If this moment isn't the start of the time line (time point 0), then the truth value of the formula has changed at this moment. For the cases that  $\varphi$  is *false* or *unknown* at  $t_3$ , similar formulations can be given.

In case we have finite variability, we also have the change pinpoint principle:

#### PROPOSITION 2.1

for all traces  $\mathcal{M}$ :

$\mathcal{M}$  satisfies finite variability  $\Rightarrow \mathcal{M}$  satisfies the change pinpoint principle

PROOF (sketch):

Suppose finite variability holds for a trace  $\mathcal{M}$ . We take a point in time  $t_3$  and we suppose  $\text{state}(\mathcal{M}, t_3) \models \varphi$ . In order to prove the change pinpoint principle for  $\mathcal{M}$ , we have to show:

$\exists t_2 \leq t_3 \forall t \in [t_2, t_3] : \text{state}(\mathcal{M}, t) \models \varphi \wedge$   
 $(t_2 \neq 0 \Rightarrow \exists t_1 < t_2 \forall t' \in [t_1, t_2) : \text{state}(\mathcal{M}, t') \not\models \varphi)$ . We call this assertion  $\star$ . So, we have to show there is period before  $t_3$  where the formula holds, and in case this period doesn't start at time 0, that there is an earlier period where the formula doesn't hold. We have two cases:

$\varphi$  holds at all time points up to  $t_3$ : In this case, we take  $t_2 = 0$ . This  $t_2$  works to validate  $\star$ .

$\varphi$  doesn't always hold before  $t_3$ : We use finite variability, with 0 substituted for  $s$  and  $t_3$  for  $t$ . This renders that the time between 0 and  $t_3$  can be divided into a number of consecutive right-open time intervals, such that the system's information state according to  $\mathcal{M}$  is stable in each interval. Time point 0 belongs to the first interval, while  $t_3$  (where  $\varphi$  holds) immediately follows the last interval. We call the left boundary of the last interval  $v$ , so this last interval is  $[v, t_3)$ . We now have two cases again, depending on whether  $\varphi$  holds in the last interval.

$\text{state}(\mathcal{M}, v) \not\models \varphi$ : This means that  $\varphi$  doesn't hold in the last interval. Thus, the interval prior to  $t_3$  where  $\varphi$  holds consists only of time point  $t_3$ . We take  $t_2 = t_3$  and  $t_1 = v$ . With this  $t_2$  and  $t_1$ ,  $\star$  holds.

$\text{state}(\mathcal{M}, v) \models \varphi$ : This means that  $\varphi$  holds in the last interval before  $t_3$ , and possibly also in earlier intervals. Because we know that  $\varphi$  holds at  $t_3$ , we can conclude that  $\varphi$  holds in the interval  $[v, t_3]$ . As  $\varphi$  doesn't hold on the entire interval  $[0, t_3]$ , there must be an earlier interval where  $\varphi$  doesn't hold. Take the most recent interval before  $v$  where  $\varphi$  doesn't hold. Suppose this interval is  $[u, w)$ . Thus in all subsequent intervals,  $\varphi$  holds, up to and inclusive  $t_3$ , that is in  $[w, t_3]$ . We take  $t_2 = w$ , and  $t_1 = u$ , and  $\star$  holds.

End of PROOF.

To express that some formula has just become true, we introduce the following notation pronounced as *just*. In the definitions below, as in the rest of this thesis, we will use ' $\equiv$ ' to mean 'is defined as'.

$$\oplus \text{state}(\mathcal{M}, t_1, \text{interface}) \models \varphi \equiv \text{state}(\mathcal{M}, t_1, \text{interface}) \models \varphi \wedge (t_1 \neq 0 \Rightarrow \exists t_2 < t_1 \forall t \in [t_2, t_1) : \text{state}(\mathcal{M}, t, \text{interface}) \not\models \varphi)$$

$$\oplus \text{state}(\mathcal{M}, t_1, \text{interface}) \not\models \varphi \equiv \text{state}(\mathcal{M}, t_1, \text{interface}) \not\models \varphi \wedge (t_1 \neq 0 \Rightarrow \exists t_2 < t_1 \forall t \in [t_2, t_1) : \text{state}(\mathcal{M}, t, \text{interface}) \models \varphi)$$

So, if a formula  $\varphi$  holds initially, we have  $\oplus \text{state}(\mathcal{M}, 0, \text{interface}) \models \varphi$ , and at all other time points  $t \neq 0$  we have  $\oplus \text{state}(\mathcal{M}, t, \text{interface}) \models \varphi$  if the truth value of  $\varphi$  changes to *true* at  $t$ .

## 2.5 Criteria for beliefs, desires and intentions

In this section, we provide formal criteria that temporal formulae from TL must adhere to in order to qualify as a representation of belief, desire or intention, respectively. We assume there is one agent on which we focus, and whose identifier is `Ag`. The criteria take a candidate formula  $\psi(\mathcal{M}, t)$ , and check whether this formula has the properties of belief, desire or intention informally described in Section 2.3. If so, then  $\psi(\mathcal{M}, t)$  is a representation of the intentional notion at hand being present in the agent in trace  $\mathcal{M}$  at time  $t$ . All formulae from TL can be tested with the criteria. For example, we could have a formula  $\varphi(\mathcal{M}, t)$  describing input- and output behaviour of an agent before and after  $t$  and test whether this formula qualifies as an intention representation. But if so, this formula is not very useful, because if we later want to know whether at some time  $t$  we can attribute intention to the agent, we have to know future information states of the agent in order to check whether  $\varphi(\mathcal{M}, t)$  is true, and so we have to wait a while before we can decide. We will focus on two categories of candidate formulae for belief, desire or intention being present at  $t$ , namely formulae describing the input history of the agent up and until  $t$  (recall the Determinism Assumption) and formulae phrasing a property of the internal agent state at time  $t$ . The latter formulae can be tested to find out whether the intentional notions an agent uses internally are proper. We discuss both classes of formulae in Section 2.5.5.

### 2.5.1 Beliefs

Recall that in Section 2.3 we have assumed that an agent believes all information which it has received in the past and which is not overridden by more recent information. Before giving a temporal characterisation of the notion of belief, we need an auxiliary definition:

DEFINITION 2.1 (Informed)

Let  $\varphi \in \text{SL}(\text{EWOnt})$ , then:

$$\begin{aligned} \text{Informed}(\varphi, t, \mathcal{M}) &\equiv \oplus \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models \text{observation\_result}(\varphi) \vee \\ &\exists \text{B} \in \text{AGENT} : \oplus \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models \text{communicated\_by}(\varphi, \text{B}) \end{aligned}$$

Here `AGENT` is a sort for the agents names. So,  $\text{Informed}(\varphi, t, \mathcal{M})$  means that the agent has just received information that  $\varphi$  (describing world state information) is true at time point  $t$ . The following characterisation of belief is based on the assumption that an agent believes a fact if it was informed about it in the past and the fact is not contradicted by later information of the opposite.

We restrict ourselves to beliefs of formulae describing the state of the world ( $\in \text{SL}(\text{EWOnt})$ ). For  $\varphi$ , the *complementary formula*  $\sim\varphi$  is defined as  $\sim\varphi \equiv \alpha$  if  $\varphi = \neg\alpha$  and  $\sim\varphi \equiv \neg\varphi$  otherwise. We adopt the convention to denote (candidate) belief formulae by  $\beta(\mathcal{M}, t)$ , corresponding to the symbols used in Figure 2.1 (Section 2.3).

**DEFINITION 2.2 (Belief representation)**

Let  $\varphi \in \text{SL}(\text{EWOnt})$ . The temporal formula  $\beta(\mathcal{M}, t) \in \text{TL}$  is an externally grounded *belief formula* for  $\varphi$  with respect to a world description  $\mathcal{W}$  if

$$\forall \mathcal{M} \in \mathcal{W} \forall t_1 : (\beta(\mathcal{M}, t_1) \Leftrightarrow \exists t_0 \leq t_1 : [\text{Informed}(\varphi, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim\varphi, t, \mathcal{M})])$$

Formulae satisfying the condition in Definition 2.2 are called externally grounded, because they only hold at  $t$  in the traces of  $\mathcal{W}$  when the externally observable agent behaviour (that is, what takes place in the input- and output interface of the agent) has certain properties. So, belief formulae are grounded in the external agent behaviour.

A formula is an externally grounded belief (or desire or intention) formula *with respect to a certain world description*. The formula is induced from the behaviour in the world description. If this world description is representative for all possible behaviours of the agent, then the externally grounded formula can be used to ascribe the intentional notion involved in traces which were not part of the world description. Whenever the externally grounded belief, desire or intention formula holds, we can conclude that the agent has the belief, desire or intention, respectively. But there is no guarantee that this ascription is correct. It could happen that a formula which is an externally grounded belief (or intention or desire) formula with respect to  $\mathcal{W}$  holds in a new trace not part of  $\mathcal{W}$ , and the criteria for belief (or intention or desire, respectively) aren't fulfilled. It's like the famous induction example of the black swans; even if you have seen a thousand swans that are all black, you can't be certain that all swans are black. Coming across a white (or blue, for that matter) swan always stays a possibility. We will often be sloppy and leave out the phrase "with respect to a world description  $\mathcal{W}$ " when dealing with externally grounded formulae.

Note that the temporal past formula  $\text{Belief}(\varphi, t_1, \mathcal{M})$  in  $\text{PL}(\text{InOnt}, t_1)$  defined by

$$\exists t_0 \leq t_1 : [\text{Informed}(\varphi, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim\varphi, t, \mathcal{M})]$$

itself is an externally grounded belief formula for  $\varphi$ .

The following proposition gives two properties of belief formulae. In the first place, it states that all externally grounded belief formulae are *temporally equivalent*, which means that they have the same truth values in each trace in

the world description, at each moment in time. Secondly, the proposition states that a belief formula for a certain fact and a belief formula for the complementary fact never are true together.

**PROPOSITION 2.2** (Properties of belief formulae)

Let  $\varphi \in \text{SL}(\text{EWont})$  be given.

**(a)** All belief formulae for  $\varphi$  are temporally equivalent; i.e.,

if  $\beta_1(\mathcal{M}, t), \beta_2(\mathcal{M}, t) \in \text{TL}$  are two externally grounded belief formulae for  $\varphi$  with respect to  $\mathcal{W}$ , then

$$\forall \mathcal{M} \in \mathcal{W} \forall t : \beta_1(\mathcal{M}, t) \Leftrightarrow \beta_2(\mathcal{M}, t)$$

**(b)** At each time point there are no belief formulae true for complementary world state formulae; i.e., for any world state formula  $\varphi$ , if  $\beta_1(\mathcal{M}, t_1)$  is a belief formula for  $\varphi$  and  $\beta_2(\mathcal{M}, t_1)$  is a belief formula for the complementary formula  $\sim\varphi$ , then

$$\forall \mathcal{M} \in \mathcal{W} \forall t : \beta_1(\mathcal{M}, t) \Rightarrow \neg\beta_2(\mathcal{M}, t)$$

Note that temporal equivalence of two belief formulae (as in part **(a)** of the proposition) does not imply equivalence of the two in the ordinary sense. Temporal equivalence is defined with respect to the models in the world description  $\mathcal{W}$ , while for classical equivalence the two belief formulae have to have the same truth values in *all* possible models. When two formulae are temporally equivalent, then for each trace within the world description  $\mathcal{W}$ , the truth values of both formulas are the same at each moment in time.

## 2.5.2 Intentions

Motivational attitudes refer in their semantics to the future actions of the agent (see Section 2.3), so it can be expected that in the criteria for intentions also reference to future actions of the agent is made. We review our assumptions regarding intentions before formalising them in two conditions that candidate formulae for intentions have to satisfy. We assume that an agent who intends to perform an action (physical, communicative or observative) will execute the action when it believes that an opportunity ( $\alpha$ ) occurs and that when the agent performs an action ( $\theta$ ), the agent intended ( $\gamma$ ) to do that. These two assumptions result in two conditions a candidate intention formula has to satisfy, the *sufficiency condition for intention* and the *necessity condition for intention*, respectively. The sufficiency condition expresses that whenever the candidate intention is present, and the most recent world information is that the agent has the opportunity to do the action, this is sufficient for an action to occur at some later point in time. The necessity condition states that for the action

under focus to occur, it is necessary that at some earlier time point the candidate intention formula holds and the most recent information is that there is an opportunity to do the action. Again, we use the symbols introduced in Figure 2.1. An opportunity  $\alpha$  is a state formula based on the ontology of EW, formalising the conditions on the state of the world under which the action can be done. In the definition below an *action atom*  $\theta(\mathcal{M}, t)$  is an atom of the form  $\text{state}(\mathcal{M}, t, \text{output}(\text{Ag})) \models \psi$  with  $\psi$  being one of `to_be_performed(a)`, `to_be_communicated.to( $\varphi, i$ )`, or `to_be_observed( $\varphi$ )`.

**DEFINITION 2.3** (Intention representation)

Let  $\alpha \in \text{SL}(\text{EWOnt})$  be an external state formula and  $\theta(\mathcal{M}, t)$  an action atom. The temporal formula  $\gamma(\mathcal{M}, t) \in \text{TL}$  is called an *externally grounded intention formula* for action atom  $\theta(\mathcal{M}, t)$  and *opportunity*  $\alpha$  with respect to a world description  $\mathcal{W}$  if the following conditions are fulfilled:

*Sufficiency condition for intention:*

$$\begin{aligned} &\forall \mathcal{M} \in \mathcal{W} \forall t_1 : (\gamma(\mathcal{M}, t_1) \wedge \\ &\exists t_0 \leq t_1 : [\text{Informed}(\alpha, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim \alpha, t, \mathcal{M})] \\ &\quad \Rightarrow \exists t_2 \geq t_1 : \theta(\mathcal{M}, t_2)) \end{aligned}$$

*Necessity condition for intention:*

$$\begin{aligned} &\forall \mathcal{M} \in \mathcal{W} \forall t_2 : (\theta(\mathcal{M}, t_2) \Rightarrow \\ &\exists t_1 \leq t_2 : \gamma(\mathcal{M}, t_1) \wedge \\ &\exists t_0 \leq t_1 : [\text{Informed}(\alpha, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim \alpha, t, \mathcal{M})]) \end{aligned}$$

The criteria above mean that a candidate temporal formula  $\gamma(\mathcal{M}, t)$  is a proper formalisation of intention to do an action, if whenever this formula holds in a trace of the world description under consideration and the agent also has non-refuted information that there is an opportunity to do the action, it acts at a later moment (sufficiency), and also that whenever the agents performs the action in a trace of the world description, there was an earlier time at which  $\gamma(\mathcal{M}, t)$  held and the agent also had non-refuted information that there was an opportunity to do the action. As is clear in the light of the previous subsection, having non-refuted information that there is an opportunity comes down to believing that there is an opportunity.

The world description provides a behaviour sample of the agent, and is used to check the hypothesis that  $\gamma(\mathcal{M}, t)$  is an intention formula. It is possible that a candidate formula is an externally grounded intention formula with respect to a world description  $\mathcal{W}$ , but not with respect to a larger domain description  $\mathcal{W}'$ , as the larger domain description can contain traces which are counterexamples of the criteria above. To have enough information to determine whether a can-

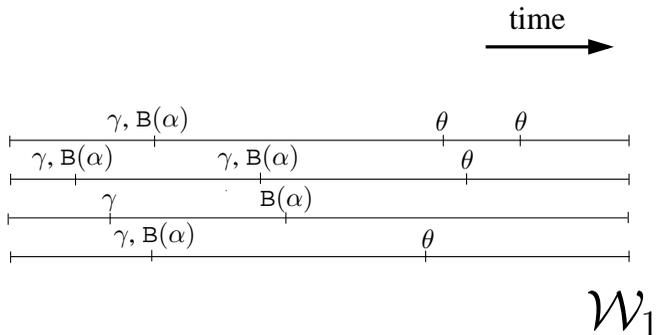


Figure 2.3: A world description

didate formula is an externally grounded intention formula that is useful for a wide range of agent behaviours, the selected world description has to be large enough and diverse enough.

Note that the criteria above don't force the intention to hold on until the moment the agent decides to do the action. In the period between the intention being present ( $t_1$ ) and the decision to do the action ( $t_2$ ),  $\gamma(\mathcal{M}, t)$  doesn't need to hold. In other words, we don't have *persistence* of motivations.

In Figure 2.3, we illustrate the intuition of the criteria for intention. In this figure, we schematically depict a world description  $\mathcal{W}_1$ , which for the purpose of the example consists of only four traces. In the previous subsection, we saw that the formula  $\exists t_0 \leq t_1 : [\text{Informed}(\alpha, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim \alpha, t, \mathcal{M})]$  is also denoted by  $\text{Belief}(\alpha, t_1, \mathcal{M})$ . For readability, we use this shorter notion and abbreviate it even further into  $\text{B}(\alpha)$ . The figure shows that at certain time points in certain traces the formulas  $\gamma$  and  $\text{B}(\alpha)$  and the action atom  $\theta$  are true. For brevity, we omit the parameters  $\mathcal{M}$  and  $t$  of these formulae. We assume that at the other time points, the formulae don't hold (or their truth value is irrelevant). To find out whether  $\gamma$  is an externally grounded intention formula, we have to check each trace to see if truth of both the candidate intention formula  $\gamma$  and the belief that there is an opportunity  $\text{B}(\alpha)$  always gives an action  $\theta$  at a later moment, and whether for each action  $\theta$  there is an earlier time at which  $\text{B}(\alpha)$  and  $\gamma$  hold. As can easily be checked, this indeed is the case, so whenever  $\gamma$  is true, the agent intends to do  $\theta$ . Note that the criteria for intention allow situations such as in the first (topmost) trace, where two action occurrences take place, even though there is only one time at which there is the combination of intention and believed opportunity. The criteria also allow that only one action takes place when there

are two earlier moments where there is the intention for  $\theta$  and the belief that there is opportunity. These things might seem strange, and indicate that it is valuable to have a large domain description (containing much more than four traces), in order to rule out unwanted intention formulas.

In contrast, in Figure 2.4,  $\gamma$  is not an externally grounded intention formula. In this figure, there is a violation of the criteria in three of the four traces. In the

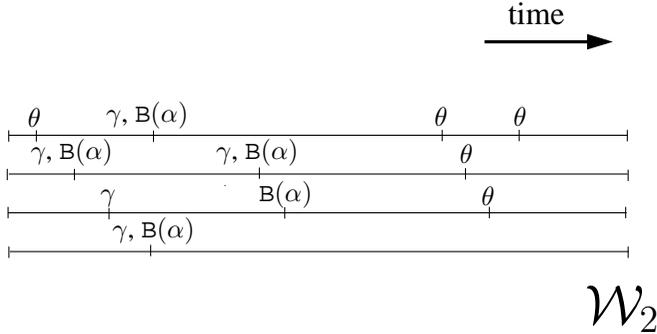


Figure 2.4: Another world description

first trace, the first occurrence of the action is not preceded by an intention together with believed opportunity. In the third trace we have the same problem, and in the fourth trace truth of the candidate intention formula and the belief that there is opportunity to do  $\theta$  don't lead to an action.

As already suggested by the examples above, externally grounded belief formulae can replace the sub-formulae formalising that there is non-refuted information on an occurrence of an occasion for the action under focus. If there is an externally grounded belief formula for the opportunity  $\alpha$  (for example  $\text{Belief}(\alpha, t, \mathcal{M})$ , as used above), then the characterisation of an intention can be reformulated.

### PROPOSITION 2.3 (Shorter intention conditions)

Let  $\alpha \in \text{SL(EWOnt)}$  be an external state formula,  $\beta_\alpha(\mathcal{M}, t)$  be an externally grounded belief formula for  $\alpha$  and  $\theta(\mathcal{M}, t)$  an action atom. The temporal formula  $\gamma(\mathcal{M}, t) \in \text{TL}$  is an externally grounded intention formula for action atom  $\theta(\mathcal{M}, t)$  and opportunity  $\alpha$  if and only if the following conditions are fulfilled:

*Sufficiency condition for intention:*

$$\forall \mathcal{M} \in \mathcal{W} \forall t_1 : (\gamma(\mathcal{M}, t_1) \wedge \beta_\alpha(\mathcal{M}, t_1) \Rightarrow \exists t_2 \geq t_1 : \theta(\mathcal{M}, t_2))$$

*Necessity condition for intention:*

$$\forall \mathcal{M} \in \mathcal{W} \forall t_2 : (\theta(\mathcal{M}, t_2) \Rightarrow \exists t_1 \leq t_2 : \gamma(\mathcal{M}, t_1) \wedge \beta_\alpha(\mathcal{M}, t_1))$$

This proposition follows immediately using Definition 2.2.

### 2.5.3 Desires

In Section 2.3, we sketched the properties of desires, which we will formalise here. An agent can desire states of the world as well as actions to be performed. Not all desires lead to intentions and subsequently to actions; the agent needs to believe an additional reason ( $\rho$ ) before it chooses to associate an intention to do an action with the desire. For each action that can contribute to the establishment of the desire, there is a different reason. For example, if you desire to have an ice-cream, and you believe that there is ice-cream in your freezer, then you probably will intend to open the freezer. But if you don't believe this, and you do believe the snack-bar on the corner sells ice-cream, you will intend to walk over to the snack-bar. A reason thus describes a state of the world in which a desire can be fulfilled by doing a certain action. Just like in the previous subsection, we have two conditions for formulae to qualify as an desire. The first is that when an agent desires ( $\delta$ ) something, and it believes the reason associated with a certain action, then it will develop the intention ( $\gamma$ ) to perform that action. And the second is that there are no intentions without desires causing them. The criteria for intention and those for desire have similar formalisations and intuitions.

DEFINITION 2.4 (Desire representation)

Let an external state formula  $\rho \in \text{SL}(\text{EWOnt})$  and an intention formula  $\gamma(\mathcal{M}, t)$  be given. The temporal formula  $\delta(\mathcal{M}, t) \in \text{TL}$  is called an *externally grounded desire formula* for intention  $\gamma(\mathcal{M}, t)$  and *reason*  $\rho$  with respect to world description  $\mathcal{W}$  if the following conditions are fulfilled:

*Sufficiency condition for desire:*

$$\begin{aligned} \forall \mathcal{M} \in \mathcal{W} \forall t_1 : (\delta(\mathcal{M}, t_1) \wedge \\ \exists t_0 \leq t_1 : [\text{Informed}(\rho, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim \rho, t, \mathcal{M})]) \\ \Rightarrow \exists t_2 \geq t_1 : \gamma(\mathcal{M}, t_2) \end{aligned}$$

*Necessity condition for desire:*

$$\begin{aligned} \forall \mathcal{M} \in \mathcal{W} \forall t_2 : (\gamma(\mathcal{M}, t_2) \Rightarrow \\ \exists t_1 \leq t_2 : \delta(\mathcal{M}, t_1) \wedge \\ \exists t_0 \leq t_1 : [\text{Informed}(\rho, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim \rho, t, \mathcal{M})]) \end{aligned}$$

The sufficiency condition for desire demands that whenever a candidate desire formula  $\delta(\mathcal{M}, t)$  holds in a trace, and there is non-refuted information on the reason  $\rho$  to pursue this desire in a certain manner, then a matching intention occurs somewhat later. The necessity condition states that whenever a certain intention arises, there must have been a matching desire and believed reason some time earlier. Again, we don't demand that the desire continues to be present until the intention is generated. Note that these criteria presuppose that intentions and desires are coupled in a one-to-one manner. Of course, this is not very realistic. For example, if I intend to visit the snack-bar, this does not necessarily mean that I desire an ice-cream. I could also have a craving for chips. In order to take care of these situations, the necessity condition can be adjusted.

As the intuitions of the criteria for desire resemble those of the criteria for intention, we only give one example to illustrate the desire criteria. In Figure 2.5, we show a world description  $\mathcal{W}_3$ . Again, we use  $\mathbb{B}(\rho)$  for  $\text{Belief}(\rho, t_1, \mathcal{M})$ . The candidate formula  $\delta$  is not an externally grounded desire representation, because in the second trace the necessity condition is violated (as there is no non-refuted information about  $\rho$ ) and in the fourth trace the sufficiency condition doesn't hold (no intention is generated).

If for the external state formulae used for the reason, any belief formula (e.g., an internal belief representation) is given, then we can reformulate the characterisation of a desire:

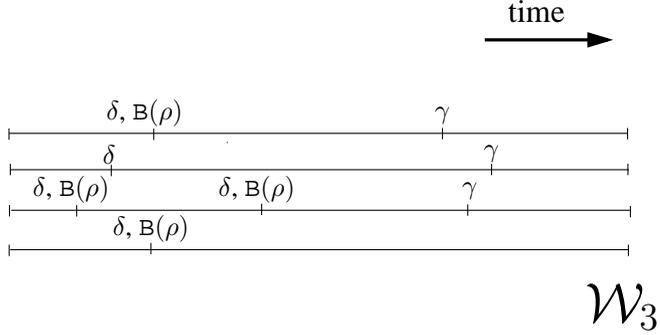


Figure 2.5: Checking a candidate desire formula

**PROPOSITION 2.4 (Shorter desire conditions)**

Let  $\rho \in \text{SL}(\text{EWOnt})$  be an external state formula,  $\beta_\rho(\mathcal{M}, t)$  an externally grounded belief formula for  $\rho$  and  $\gamma(\mathcal{M}, t)$  an intention formula. The temporal formula  $\delta(\mathcal{M}, t) \in \text{TL}$  is an externally grounded desire formula for intention  $\gamma(\mathcal{M}, t)$  and reason  $\rho$  if and only if the following conditions are fulfilled:

*Sufficiency condition for desire:*

$$\forall \mathcal{M} \in \mathcal{W} \forall t_1 : (\delta(\mathcal{M}, t_1) \wedge \beta_\rho(\mathcal{M}, t_1) \Rightarrow \exists t_2 \geq t_1 : \gamma(\mathcal{M}, t_2))$$

*Necessity condition for desire:*

$$\forall \mathcal{M} \in \mathcal{W} \forall t_2 : (\gamma(\mathcal{M}, t_2) \Rightarrow \exists t_1 \leq t_2 : \delta(\mathcal{M}, t_1) \wedge \beta_\rho(\mathcal{M}, t_1))$$

## 2.5.4 Changing the BDI-criteria

The criteria for being a proper belief, desire or intention representation given in the previous subsections embody assumptions on these notions (see also Section 2.3). There are many theories and conceptions of what belief, desire and intention are, and the formalisation of these notions can also depend on system or domain characteristics. The assumptions of Section 2.3 thus might not be applicable in all contexts. The criteria can be modified to embody different assumptions. In this subsection, we will give examples of different assumptions and show a few modified criteria.

The criterion for belief is based on the assumption that the agent believes all information it receives, as long as it is not overridden by newer information. If required, these assumptions can also be replaced by less simple ones, possibly in a domain-dependent manner. For example, we could take into account

reliability of sensory processes in observation or reliability of other agents in communication. Or we can assume that an agent will believe some fact when it received non-refuted information from two different sources (two different agents, or one agent and observation) that the fact holds. We will show a modified criterion for this last assumption. First, we need an additional notion for being informed from two different sources:

DEFINITION 2.5 (DoublyInformed)

Let  $\varphi \in \text{SL}(\text{EWOnt})$ , then:

$$\begin{aligned} \text{DoublyInformed}(\varphi, t_1, t_2, \mathcal{M}) &\equiv \\ &[\oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{Ag})) \models \text{observation\_result}(\varphi) \wedge \\ &\exists A \in \text{AGENT} : \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models \text{communicated\_by}(\varphi, A)] \vee \\ &[\exists A \in \text{AGENT} : \oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{Ag})) \models \text{communicated\_by}(\varphi, A) \wedge \\ &\quad \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models \text{observation\_result}(\varphi)] \vee \\ &[\exists A, B \in \text{AGENT} : A \neq B \wedge \\ &\quad \oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{Ag})) \models \text{communicated\_by}(\varphi, A) \wedge \\ &\quad \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models \text{communicated\_by}(\varphi, B)] \end{aligned}$$

$\text{DoublyInformed}(\varphi, t_1, t_2, \mathcal{M})$  means that agent  $\text{Ag}$  has received information regarding  $\varphi$  from two different sources, at time points  $t_1$  and  $t_2$ . Now, we can define a new belief criterion:

DEFINITION 2.6 (An alternative belief representation)

Let  $\varphi \in \text{SL}(\text{EWOnt})$ . The temporal formula  $\beta(\mathcal{M}, t) \in \text{TL}$  is an externally grounded *belief formula* for  $\varphi$  if

$$\begin{aligned} \forall \mathcal{M} \in \mathcal{W} \forall t_0 : (\beta(\mathcal{M}, t_0) \Leftrightarrow \\ \exists t_2 \leq t_0 \exists t_1 < t_2 : [\text{DoublyInformed}(\varphi, t_1, t_2, \mathcal{M}) \wedge \\ \forall t \in [t_1, t_0] : \neg \text{Informed}(\sim \varphi, t, \mathcal{M})]) \end{aligned}$$

So, using this alternative criterion a candidate belief formula is a good belief representation for  $\varphi$  if it is temporally equivalent with being informed from two different sources about  $\varphi$ , while not receiving information on  $\varphi$  not holding since the information from the first source came in.

For another variant of the belief criterion, we look at the time between being informed and the belief being formed. With the original criterion, this time is zero; the moment the agent receives information regarding  $\varphi$ , it believes this information, until it is refuted. It is very reasonable that an agent needs time to process incoming communication and observations, before it creates new beliefs. To allow for this time, we modify the original belief criterion. If

the maximum time the agent needs to process incoming information into new beliefs is  $\epsilon$ , we have the following alternative conditions for belief:

**DEFINITION 2.7** (Yet another belief representation)

Let  $\varphi \in \text{SL}(\text{EWOnt})$ . The temporal formula  $\beta(\mathcal{M}, t) \in \text{TL}$  is an externally grounded *belief formula* for  $\varphi$  if the following conditions are fulfilled:

*Sufficiency condition for belief:*

$$\forall \mathcal{M} \in \mathcal{W} \forall t_1 : ([\exists t_0 \leq t_1 - \epsilon : \text{Informed}(\varphi, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1] : \neg \text{Informed}(\sim \varphi, t, \mathcal{M})] \Rightarrow \beta(\mathcal{M}, t_1))$$

*Necessity condition for belief:*

$$\forall \mathcal{M} \in \mathcal{W} \forall t_1 : (\beta(\mathcal{M}, t_1) \Rightarrow \exists t_0 \leq t_1 : [\text{Informed}(\varphi, t_0, \mathcal{M}) \wedge \forall t \in [t_0, t_1 - \epsilon] : \neg \text{Informed}(\sim \varphi, t, \mathcal{M})])$$

The first condition formalises the intuitive idea that whenever the agent was informed about  $\varphi$  at least  $\epsilon$  time ago, and this information hasn't been refuted until now, then the agent believes  $\varphi$ . This condition is asymmetric, in the sense that  $\epsilon$  only plays a role in the first conjunct of the lefthandside of the implication, and not in the second. This is so because  $\epsilon$  is the *maximum* reasoning time the agent needs. Sometimes, the agent is quicker. So, we have to exclude refuting information in the entire interval  $[t_0, t_1]$ , and not only in  $[t_0, t_1 - \epsilon]$ . The second condition has a similar asymmetry. It states that whenever the agent believes  $\varphi$ , it must have received information on  $\varphi$  earlier, which wasn't refuted until  $\epsilon$  time ago.

The criteria for intentions and desire also can be adjusted to accommodate for different intuitive conceptions of intention and desire. The criteria for intention as given in Definition 2.3 formalise the case that all actions are intended actions. However, it is not difficult to define weaker variants. For example, if also unintended actions are allowed, the second (necessity) condition can be left out. Weaker desire notions can be defined as well. For example, the second condition of Definition 2.4, that no intentions occur without desire, may be debatable. If also undesired intentions are allowed, the second (necessity) condition of Definition 2.4 should be dropped.

Another possible adjustment of the criteria for intention and desire concerns *persistence*, which means that the reasons for the rise of an attributed notion must continue to be valid as long as the notion is present. Thus, the desire and the believed reason that gave rise to a later intention must stay present until the intention dissolves, and the intention to do an action and its believed opportunity must continue to be true until the action has been initiated. Alternative desire criteria and intention criteria can be found to incorporate this.

## 2.5.5 Ascription and checking

The criteria for being a belief-, intention- or desire representation can be applied to all kinds of formulae from TL. To give a far-fetched example, in case we have a single agent system with a cleaning robot in an office environment, we could phrase a candidate formula for the robot intending to clean a window at  $t_1$  stating that the window is dirty at  $t_1$  and that it is clean at some time  $t_3 > t_1$ . If the only way the window can become clean is the robot cleaning it, this formula is an externally grounded intention formula for the action of cleaning windows. But this formula is not very useful. The agent can't have used this formula in its reasoning to decide to clean the window, because the formula refers to the future, and the agent can't reason using information that is not yet available. If we want to use the formula to attribute intention to the agent, we have the same problem; in order to decide whether the agent intends to clean the window at  $t_1$ , we have to wait till after the agent has actually cleaned the window, and only then we can attribute the intention to the agent. It would be much more useful if we find a past formula describing the observable behaviour of the agent before  $t_1$ .

Actually, there are two categories of temporal formulae that we can sensibly consider candidate BDI representations. The first category is formed by the formulae  $\psi(\mathcal{M}, t) \in \text{PL}(\text{InOnt}, t)$ , which are formulae describing the input history of the agent up and until  $t$ . In case the Determinism Assumption holds, the history of an agent's input interface completely determines its behaviour. So, these formulae are useful candidates for being a representation of an intentional notion. If such a past formula qualifies as a BDI representation, then we can *attribute* or *ascribe* the notion at hand to the agent based on the past input behaviour of the agent. When we build this attribution quality into another agent in an agent system, this agent can understand and predict the actions of the agent studied, without having to wait until the agent actually has acted. This can be very useful. Ascription of intentional notions is possible irrespective of whether the agent in focus employs BDI-notions for its internal reasoning. If the agent uses some form of BDI internally, we ignore this in the attribution process.

However, if *internal* notions of belief, desire and intention happen to exist in an agent, or at least are claimed (e.g., by the designer of the agent), our framework can be applied to them as well. So, for the other category of candidate BDI formulae, we focus at agents that internally use BDI notions for their decision making. Using the criteria, we can check whether the notions used internally are proper formalisations of BDI. In this case, the category of formulae which are subjected to the criteria contains formulae of the form

$\text{state}(\mathcal{M}, t, \text{internal}(\text{Ag})) \models \sigma$ , where  $\sigma \in \text{SL}(\text{InternalOnt})$  is an internal state formula formalising belief, desire or intention. If we apply the criteria to these kind of formulae, we are *checking* whether the intentional notions the agent internally uses are right, in the sense that they live up to the demands to be a proper BDI representation.

We introduce special terminology for formulae from the two categories above that fulfil the criteria:

**DEFINITION 2.8** (Historical external BDI representations)

- \* The past formula  $\beta(\mathcal{M}, t) \in \text{PL}(\text{InOnt}, t)$  is called a *historical external belief representation* for  $\varphi$  if  $\beta(\mathcal{M}, t)$  is an externally grounded belief formula for  $\varphi$ .
- \* The past formula  $\gamma(\mathcal{M}, t) \in \text{PL}(\text{InOnt}, t)$  is called a *historical external intention representation* for action atom  $\theta(\mathcal{M}, t)$  and opportunity  $\alpha$  if  $\gamma(\mathcal{M}, t)$  is an externally grounded intention formula for  $\theta(\mathcal{M}, t)$  and opportunity  $\alpha$ .
- \* The past formula  $\delta(\mathcal{M}, t) \in \text{PL}(\text{InOnt}, t)$  is called a *historical external desire representation* for intention  $\gamma(\mathcal{M}, t)$  and reason  $\rho$  if it is an externally grounded desire formula for intention  $\gamma(\mathcal{M}, t)$  and reason  $\rho$ .

**DEFINITION 2.9** (Internal BDI representations)

- \* The internal state formula  $\beta \in \text{SL}(\text{InternalOnt})$  is called an externally grounded *internal belief representation* for  $\varphi$  if the formula  $\text{state}(\mathcal{M}, t, \text{internal}(\text{Ag})) \models \beta$  is an externally grounded belief formula for  $\varphi$ .
- \* The internal state formula  $\gamma \in \text{SL}(\text{InternalOnt})$  is called an externally grounded *internal intention representation* for action formula  $\theta(\mathcal{M}, t)$  and opportunity  $\alpha$  if the formula  $\text{state}(\mathcal{M}, t, \text{internal}(\text{Ag})) \models \gamma$  is an externally grounded intention formula for  $\theta(\mathcal{M}, t)$  and opportunity  $\alpha$ .
- \* The internal state formula  $\delta \in \text{SL}(\text{InternalOnt})$  is called an externally grounded *internal desire representation* for intention  $\gamma(\mathcal{M}, t)$  and reason  $\rho$  if the formula  $\text{state}(\mathcal{M}, t, \text{internal}(\text{Ag})) \models \delta$  is an externally grounded desire formula for intention  $\gamma(\mathcal{M}, t)$  and reason  $\rho$ .

## 2.5.6 Example

The following simple example illustrates the attribution process. The agent in this example is an artificial mouse, that is in an environment where there some-

times is food. The mouse agent is sometimes prevented from getting to the food by a transparent screen. We have a world description  $\mathcal{W}$  of the system, with four rather short traces. Of course, for reliable attribution of mental attitudes we need a lot more than four traces, but this small set suffices to illustrate the idea. The mouse receives observation results on the presence of the screen and the food. This mouse doesn't communicate and doesn't receive communication from other agents. Depending on the circumstances it can decide to eat the food (action eat). Below we display the world description  $\mathcal{W}$ . The traces concern the input- and output interface of the agent. For shortness, we abbreviated the atoms present in the interfaces. Instead of `observation_result(food)` we simply use `food`, and we don't mention that this atom is part of the *input* interface of the agent. Similarly,  $\neg$ screen denotes that `observation_result( $\neg$ screen)` is present in the input interface of the mouse, and `eat` denotes that the output interface contains the action atom `to_be_performed(eat)`.

<i>time</i> <i>trace</i>	<i>time</i> <i>point 0</i>	<i>time</i> <i>point 1</i>	<i>time</i> <i>point 2</i>	<i>time</i> <i>point 3</i>	<i>time</i> <i>point 4</i>	<i>time</i> <i>point 5</i>
<i>trace 1</i>	food screen	$\neg$ food $\neg$ screen	food screen	food $\neg$ screen	food $\neg$ screen eat	food $\neg$ screen eat
<i>trace 2</i>	$\neg$ food $\neg$ screen	food $\neg$ screen	$\neg$ food screen	food $\neg$ screen	food $\neg$ screen	food $\neg$ screen eat
<i>trace 3</i>	$\neg$ food $\neg$ screen	$\neg$ food $\neg$ screen	food screen	food $\neg$ screen	food $\neg$ screen	food $\neg$ screen
<i>trace 4</i>	food $\neg$ screen	$\neg$ food $\neg$ screen	food screen	food screen	food $\neg$ screen	food $\neg$ screen eat

Table 2.1: Example set of observed traces

Because this mouse doesn't receive any messages, being informed is equivalent with receiving observation results. It can easily be checked that the formula

$$\beta_{\varphi}(\mathcal{M}, t_1) \equiv$$

$$\exists t_0 \leq t_1 : \oplus \text{state}(\mathcal{M}, t_0, \text{input}(\text{mouse})) \models \text{observation\_result}(\varphi) \wedge$$

$$\forall t \in [t_0, t_1] : \neg \oplus \text{state}(\mathcal{M}, t, \text{input}(\text{mouse})) \models \text{observation\_result}(\sim \varphi)$$

is an externally grounded belief formula for  $\varphi$  according to the belief criterion in Definition 2.2.

For the state formula  $\neg\text{screen}$  as opportunity, the following past formula is an adequate intention representation (with regard to the `eat`-action):

$$\begin{aligned} \gamma(\mathcal{M}, t) \equiv & \text{state}(\mathcal{M}, t, \text{input}(\text{mouse})) \models \text{observation\_result}(\text{food}) \wedge \\ & \exists t_1 \leq t : [\text{state}(\mathcal{M}, t_1, \text{input}(\text{mouse})) \models \text{observation\_result}(\neg\text{food}) \wedge \\ & \exists t_2 \leq t_1 : \text{state}(\mathcal{M}, t_2, \text{input}(\text{mouse})) \models \text{observation\_result}(\text{food})] \end{aligned}$$

Informally this formula can be explained as follows: the agent has the intention to eat at each time point that food is visible and in the past the agent experienced that visible food can suddenly disappear. Checking that this formula obeys the intention criteria is easy; the two conditions from Definition 2.3 must hold for the world description depicted above, with  $\neg\text{screen}$  substituted for  $\alpha$  and  $\text{state}(\mathcal{M}, t, \text{output}(\text{Ag})) \models \text{to\_be\_performed}(\text{eat})$  substituted for  $\theta(\mathcal{M}, t)$ . For this specific example, the criteria state that the mouse decides to eat if and only if at some earlier time point the formula above held (meaning that the agent has seen that there is food, which can disappear) and the mouse also believed that there was no screen. As can be seen from the traces in Table 2.1, this is indeed the case.

## 2.6 Anticipatory reasoning in organisations

Viewed from a dynamic perspective, organisational structure (cf. [43]) provides specifications of constraints on role behaviour and interactions (cf. [44, 45]). These specifications to a certain extent enforce coordinated dynamics on an organisation. In human organisations role specifications usually do not completely prescribe behaviours, however. To a greater or lesser extent some space of freedom in behaviour and personal initiative is allowed. This freedom has its positive elements; in the first place, human agents can find more satisfaction if they can do things in their own way. In the second place an organisational structure does not anticipate on all possible circumstances; in unforeseen situations it can be beneficial if agents have some space to improvise.

The reverse of the medal, however, is that this freedom also may provide possibilities to agents to avoid certain behaviours (based on their individual interest) as expected by others, and thus may decrease the extent of coordination. To function more efficiently in an organisation, where roles do not completely prescribe behaviour, it is useful if agents fulfilling a certain role in the organisation can reason in an anticipatory sense about the behaviour of the agents in other roles, for example, using the intentional stance. This section addresses this application of the framework introduced in Section 2.5 in more detail. Some examples of the phenomena described for human organisations are:

- (a) An employee has done something very important very wrong, and deliberates whether or not to tell his manager: *'If he believes that I am the cause of the problems, he will try to fire me.'*
- (b) An employee has encountered a recurring problem, and knows a solution for this problem, on which he would like to work. He deliberates about how to propose this solution to his manager. *'If I tell this solution immediately he will not believe that the problem is worth working on it. If I make him aware of the problem, and do not tell a solution, he only will start to think about it himself for a while, without finding a solution, and then forget about it. If I make him aware of the problem and give some hints that direct him to a (my) solution, he will believe he contributed to a solution himself and he will want me to work on it.'*
- (c) A manager observes that a specific employee in the majority of cases functions quite cooperatively, but shows avoidance behaviour in other cases. In these latter cases, the employee starts trying to reject the task if he believes that his agenda already was full-booked for the short term, he believes there are colleagues capable of doing the task, and he believes there are colleagues available with less full-booked agendas. Further observation by the manager reveals the pattern that the employee shows avoidance behaviour, in particular, in cases that a task is only asked shortly before its deadline, without the possibility to anticipate on the possibility of having the task allocated. The manager deliberates about this as follows: *'If I know beforehand the possibility that a last-minute task will occur, I can tell him the possibility in advance, and in addition point out that I need his unique expertise for the task, in order to avoid the behaviour that he tries to avoid the task when it actually comes up.'*

The reasoning processes on predicted behaviours described in (a) to (c) can be based on prescribed role behaviours (as may be the case in (a)), or on an analysis of the other agent's personal motivations (as is the case in (b) and (c)). Especially in these latter cases, the analysis framework developed in this chapter is applicable. To show this, we address example (c) by making the following interpretation.

The *desire* to avoid a task is created at time  $t$  by the employee if the following holds for the history:

- \* at time  $t$  the employee heard the request to perform the task
- \* at time  $t$  the employee observes that the task has to be finished soon

- \* the employee did not hear of the possibility of the task at any earlier time point

The *intention* to avoid a task is generated at time  $t$  if the following holds for the history:

- \* the desire to avoid the task is available at time  $t$
- \* the belief that colleagues are capable of doing the task is available at time  $t$
- \* the belief that colleagues are not full-booked is available at time  $t$

The *action* to avoid the task is generated at time  $t$  if the following holds for the history:

- \* the intention to avoid the task is available at time  $t$
- \* the belief that the employee's own agenda is full-booked is available at time  $t$

The formalisations of these conditions are as follows:

- \* The *input ontology* InOnt of the employee agent includes:

observation\_result(task\_urgent),  
 observation\_result(own\_agenda\_full),  
 observation\_result(colleagues\_agenda\_not\_full),  
 observation\_result(colleagues\_capable\_of\_task),  
 communicated\_by(task\_request, manager),  
 communicated\_by(task\_possibility, manager).

- \* The *output ontology* OutOnt includes:

to\_be\_communicated\_to(task\_rejection, manager).

Define the past formula  $\delta(\mathcal{M}, t) \in \text{PL}(\text{InOnt}, t)$  for the *desire* to avoid the task by  $\text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models \text{communicated\_by}(\text{task\_request}, \text{manager}) \wedge \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models \text{observation\_result}(\text{task\_urgent}) \wedge \neg \exists t_0 < t : \text{state}(\mathcal{M}, t_0, \text{input}(\text{Ag})) \models \text{communicated\_by}(\text{task\_possibility}, \text{manager})$

The *reason*  $\rho$  to generate an avoidance intention is:

colleagues\_agenda\_not\_full  $\wedge$  colleagues\_capable\_of\_task

The past formula  $\gamma(\mathcal{M}, t) \in \text{PL}(\text{InOnt}, t)$  for the *intention* to avoid the task is

defined in short form by

$$\delta(\mathcal{M}, t) \wedge \text{Belief}(\text{colleagues\_agenda\_not\_full} \wedge \text{colleagues\_capable\_of\_task}, t, \mathcal{M})$$

The extensive form is:

$$\begin{aligned} & \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models \text{communicated\_by}(\text{task\_request}, \text{manager}) \wedge \\ & \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models \text{observation\_result}(\text{task\_urgent}) \wedge \\ & \neg \exists t_0 < t : \text{state}(\mathcal{M}, t_0, \text{input}(\text{Ag})) \models \\ & \quad \text{communicated\_by}(\text{task\_possibility}, \text{manager}) \wedge \\ & \exists t_2 \leq t : [\text{Informed}(\text{colleagues\_agenda\_not\_full} \wedge \\ & \quad \text{colleagues\_capable\_of\_task}, t_2, \mathcal{M}) \wedge \\ & \forall t_1 \in [t_2, t] : \neg \text{Informed}(\sim (\text{colleagues\_agenda\_not\_full} \wedge \\ & \quad \text{colleagues\_capable\_of\_task}), t_1, \mathcal{M})] \end{aligned}$$

The *opportunity*  $\alpha$  to perform the avoidance action is:

$$\text{own\_agenda\_full}$$

The past formula  $\omega(\mathcal{M}, t) \in \text{PL}(\text{InOnt}, t)$  for the *action* to avoid the task is defined in its short form by

$$\gamma(\mathcal{M}, t) \wedge \text{Belief}(\text{own\_agenda\_full}, t, \mathcal{M})$$

Note that the desire to avoid a task immediately springs up when the employee hears of the urgent task, and that the intention to avoid the task and the avoidance action itself are also instantly generated when the agent has the relevant beliefs. For human agents, with their quick brains, this seems plausible. The criteria for motivational attitudes that we presented in Section 2.5 allow a delay for reasoning, which this employee agent apparently doesn't need.

Given this formalisation it can be illustrated how the manager agent can reason and act in an anticipatory manner to avoid the employee's avoidance desire, intention and/or action to occur. This can be done in the following three manners:

**(1) Avoiding the desire to occur**

This can be obtained by communicating in advance to the employee that possibly a last minute task will occur. This would make the third condition in the definition of the temporal desire formula fail.

**(2) Avoiding the intention to occur (given that the desire occurs)**

This can be obtained by refutation of the reason to generate the intention, e.g., by telling the employee that he is the only one with the required expertise.

**(3) Avoiding the action to occur (given that the intention occurs)**

This can be obtained by refutation of the opportunity, e.g., by taking one of the (perhaps less interesting) tasks from his agenda and re-allocating it to a colleague.

## 2.7 Discussion

In this chapter, we introduced formal criteria for intentional notions. These criteria relate behaviour patterns of agents to informational and motivational attitudes. Viewed from the outside, the behaviour of an agent is a complex function, which selects actions to be performed (physical, communicative or observative), based on the history of received information (by observation or communication) and earlier performed actions. In our view, intentional notions can be used to abstract the interaction histories into intuitive and compact representations. This way, mental states are grounded in interaction histories. In many other approaches using BDI notions (logics, programming languages and agent architectures), this relation between beliefs, desires and intentions and the observations and actions of the agent is less clear. A contribution of the work in this chapter is thus that BDI notions that satisfy the conditions given in Section 2.5 have a rigorously defined meaning in terms of agent behaviour, and can't be used arbitrarily. This doesn't imply that we think our conceptions on BDI as laid down in the criteria for BDI are the right ones; the criteria can be adjusted to accommodate other definitions.

Basically, the criteria can be used in three manners. The criteria allow for (1) externally ascribing motivational attitudes to agents (that may not use any belief, desire or intention internally) by defining these notions in terms of the external behaviour of the agent, (2) for analysis of internal notions (in case the agent possesses these), to verify that these internal notions have a well defined relation with the behaviour of the agent, and (3) for anticipatory reasoning to affect the circumstances that may lead to the generation of beliefs, desires and/or intentions. The combination of (1) and (3) allows an agent to interpret the behaviour of other agents. Within a multi-agent system, an agent can observe the behaviour of other agents, attribute justified intentional notions to their behaviour, and (partially) predict the behaviour of the other agents. For example, an agent responsible for the safe use of a car might learn to predict that a human standing on the edge of the pavement and looking across the road is probably intending to cross the road.

The application of our formalisation of intentional dynamics in anticipatory reasoning (and acting) makes it useful to analyse certain phenomena occurring in organisations. As shown by a number of examples in Section 2.6, usually

organisations leave some freedom in performing a certain role. To cooperate with other agents with such freedom, agents within an organisation not seldom try to affect, in an anticipatory manner, the circumstances that may lead to the generation of other agent's beliefs, desires and intentions. The capability of performing such anticipatory reasoning and acting may be crucial within organisations (where often some of the agents have certain 'directions for use'). To avoid unnecessary obstruction of the organisation's processes, these 'directions for use' better can be taken into account in cooperation. Section 2.6 shows in detail how this can be done based on the framework introduced in Section 2.5. Based on a number of experiences (observed traces) a temporal representation can be identified; this may be a computationally expensive process which has to be performed once (for example, off-line). After such a representation has been identified, it can be reused in a very efficient manner in all relevant new situations the agent encounters (on-line).

A second type of application of this work can be found in verification of agents internally designed on the basis of a BDI-model. The criteria presented in this chapter can be used to verify whether an internal representation, meant to represent some intentional notion, is a correct formalisation of such an intentional notion.

As a third use from an application-oriented perspective, the results presented in this chapter are relevant for Requirements Engineering for distributed and agent systems. Requirements for agents often concern agent behaviour; analysis and specification of such requirements is a difficult process. In practice, specification of requirements for simple reactive behaviour is feasible, but if the behaviours become more complex, requirements specification becomes much harder [23, 37, 57, 82]. In order to tackle this complexity, abstraction of complex behaviour patterns into concise notions can be very helpful. The importance of using more abstract notions in requirements specification, as opposed to the more directly formulated behaviour constraints, is stressed in [23]. Ideally, to support reuse of agents, the aim is to specify behavioural requirements without any reference or commitment to the internal structures or states of the agent. However, in practice, when specifying more complex behaviour, often not only reference is made to the dynamics of input and output states of the agent, but also to internal states, as these may contain the sought-for abstract notions. This may obstruct replacement and reuse of agents; if another agent is introduced it may have a different internal structure. One possible solution for this problem is to restrict reuse to agents with some comparable unified standard internal structure, for example a standardised BDI-structure.

The solution that can be proposed on the basis of this chapter is a different one. It is shown that to be able to use high level concepts in specification of

behavioural requirements, it is not necessary that the agent actually possesses these concepts. We showed how these concepts can also be attributed from outside, and still have a formal definition in terms of the input and output states, as required within a principled Requirements Engineering process. This combines the best of two worlds: (1) requirements specification at a higher level of abstraction, and (2) not demanding a specific internal structure within the agents. In addition, if agents do happen to possess such notions, it is possible to relate them to the concepts introduced here, as indicated. An added benefit of using abstract intentional notions for formulating requirements is that they are intuitively clear to all parties involved. Often, the stakeholders of the system to be developed, the requirements engineers, and the system designers do not use the same concepts while talking about the intended behaviour of the system. Intentional notions are easier to understand by all because of their intuitive meaning.

In order to show that the defined notions and criteria for BDI are operational and provide a basis to develop applications of agents that monitor and interpret the behaviour of other agents, we developed a dedicated agent architecture for intention attribution and built an executable prototype implementation. In [76], we report on this work.

The formal analysis and implementation of belief, desire and intentions presented here differs from the approaches in BDI logics [21, 85, 102] in that we relate intrinsically internal notions to external notions, like observations, communications and actions. We do this taking into account the dynamic nature of the ongoing interaction of agents with their environment. Thus, the motivational criteria in Section 2.5 relate the occurrence of intentional notions to temporal patterns in the externally observable agent behaviour. In BDI logics, the static connections (at one specific moment in time) between belief, desire and intention generally are formalised through axioms, but the dynamic connection between motivations and behaviour is neglected.

The temporal trace language TL used in our approach is much more expressive than standard temporal logics in a number of respects. In the first place, it has *order-sorted predicate logic* expressivity, whereas most standard temporal logics are propositional. Secondly, the explicit reference to *time points and time durations* offers the possibility of modelling the dynamics of real-time phenomena, such as sensory and neural activity patterns in relation to mental properties (cf. [100]). Third, in our approach states are *three-valued*; the standard temporal logics are based on two-valued states, which implies that for a given trace a form of closed world assumption is imposed. For example, in Concurrent METATEM [47], if the executable temporal logic specification leaves some atom unspecified, during construction of a trace the semantics will

force it to be false. To avoid this, an atom has to be split into a positive and a negative variant. Fourth, the possibility to quantify over traces allows for specification of *more complex behaviours*. As within most temporal logics, reactivity and proactiveness properties can be specified. In addition, in our language also properties of different types of adaptive behaviour can be expressed. For example, we can formalise the property ‘exercise improves skill’, which involves the comparison of two alternatives for the history. In TL, traces (that is, histories) are part of the language, and thus we can compare different traces and even quantify over traces. In linear time temporal logics different alternative histories cannot be compared. Branching time temporal logics do allow comparing different histories. Fifth, in our language it is possible to define *local languages* for parts of a system. For example, the distinction between internal, external and interface languages is crucial, and is supported by the language, which also entails the possibility to quantify over system parts; this allows for specification of system modification over time. Sixth, since state properties are used as first class citizens in the temporal trace language, it is possible to explicitly refer to them, and to quantify over them, enabling the specification of what are sometimes called *second-order properties*, which are used in part of the philosophical literature (e.g., [79]) to express functional roles related to mental properties or states. In this chapter only part of the features of the language as discussed above are exploited. But the benefit of using this expressive language is that we can extend our approach to more complex behaviours and mental properties, such as, for example, relative adaptive behaviours.

An approach that in some aspects is similar in perspective to ours, is that of Rosenschein and Kaelbling [107]. They ascribe knowledge to so-called situated automata, which are processes that do not have any internal representation of knowledge. A process with a certain internal state  $S$  knows  $\varphi$  if  $\varphi$  is true in all environment situations which are possible when the process is in state  $S$ . Our approach for ascribing beliefs is different; we relate belief to the acquired information on the environment. Furthermore, Rosenschein and Kaelbling give no account of desire and intention, which is a main contribution of our work. The same holds for recent work presented in [132], where Wooldridge and Lomuscio concentrate on the informational aspects, and abstract from motivational and temporal aspects; actually, in [132] exploration of the temporal aspects, as presented above, is mentioned as one of the four items on the list of issues for future work.

In research on plan recognition, such as [3, 81, 99], ascription of intentional notions is also done. The observing agent ascribes intentions and plans that are probable to the actor, based on observed actions of the actor agent. Plan recognition is performed using data on the actions from a single, ongoing interaction

of the agent, and uses domain knowledge on actions and their expected effects in a crucial manner. Our approach is quite different. The analysing agent primarily takes into account circumstances that may lead to certain intentions, beliefs or desires, using information on the observed and communicated information in the past of the actor studied, in order to find hypothetical past formulas representing the beliefs, desires and intentions of this agent. These formulas are tested against information on the behaviour of the observed agent during a significant number of (possible) interactions of the agent. No domain knowledge on actions and effects is used. Once an externally grounded intentional formula has been found, it is possible again and again to anticipate on the generation of intentional attitudes at forehand, without any action being performed by the actor.

From a fundamental philosophical perspective the approach presented here provides a formalisation of philosophical views on the explanation of behaviour, as addressed informally in, e.g. [31]. The characterisations introduced provide a formally defined bridge between an agent's mentalistic notions such as beliefs, desires and intentions, and materialistic notions such as observation and action performance in the world. Our approach has its perspective on grounding of mental states in the interaction in common with [9, 19]: in particular the relation between the internal agent state and interaction with the environment in the past on the one hand and potential further interactions in the future on the other hand is similar to our views (see the citation in the introduction above). Also in [20] emphasis is put on the functioning of cognition in interaction with the environment. A difference is that in our approach a formalisation is proposed, and that an explicit relation of the interaction patterns with a number of well-known intentional (BDI) notions is addressed. In continuance to the work in this chapter, [71] introduces a formalisation of the ideas in [9]. Another difference with the philosophical efforts mentioned above is that in our approach no commitments to specific internal (goal-directed) system structures and specific internal states need to be made.

The work of Wijngaards, Treur and Jonker [77] sprouts from the same inspiration and intuition as our work. In [77], mind and matter aspects of the dynamics of intentional behaviour are analysed and simulated using an implemented software environment. We restrict our attention to analysis in this chapter. Another difference is that Wijngaards, Treur and Jonker presuppose internal motivational attitudes in the agent, while we do not. The temporal relationships between external behaviour and intentional notions in our work and [77] are slightly different, but similar in spirit.

In this chapter, we provided means to abstract the complex interactions that agents have with their environment into intuitive, compact intentional no-

tions. Anthropomorphic notions aid in making agent systems graspable and designable, and are thus at the heart of agent technology.

---

## Reuse and Abstraction in Verification: Agents Interacting with Dynamic Environments

---

*Oh, baby, baby, it's a wild world  
I'll always remember you like a child, girl*  
Cat Stevens

### 3.1 Introduction

Agents sometimes are called the next software paradigm [66, 68, 128], as the anthropomorphically inspired agent concepts provide a new manner to conceive computer systems. Agents in multi-agent systems generally *interact* with each other and with their environment in different ways than other software entities do. Comparing agents to objects, agents are autonomous, which means they have total control over their own state and behaviour, while objects have to react to requests from other objects for certain actions (method invocation) by providing the services requested. Agents can always refuse, or not react

at all. Comparing agents to imperative programs, the gap is even bigger. Imperative programs usually take an input from the user, and process this into a result. During the computation, the program isn't open for unrequested information coming in from its environment; the program can perform additional information seeking, but then the initiative is taken by the program. Flexible communication between processes and interaction with the dynamic world outside the imperative program is mostly not present.

The four discerning characteristics of agents according to Wooldridge and Jennings in [130] are *autonomy* (agents don't need intervention of humans or of controlling software processes in order to operate), *social ability* (agents interact with each other through communication and coordination), *reactivity* (agents perceive their environment and respond to changes) and *pro-activeness* (agents take the initiative in order to pursue their objectives). The combination of these four agent characteristics leads to agents that display complex interactive behaviour in the environment consisting of the other agents and the world the agents inhabit.

For agent-oriented software development to become a mature Software Engineering paradigm, the scientific community needs to create principled methods to develop agent-based systems. These methods should provide for the whole software life-cycle, consisting of requirements analysis, specification, design, implementation, verification and/or testing. Of these phases in the development process of software, verification seems to be getting the least attention in the agent research community (exceptions exist; we mention the work of Fisher and Wooldridge [48]). Currently, there are many agent architectures, such as IRMA [13], GRATE [67], INTERRAP [93] and PRS [50] (see also [92] for an overview of agent architectures). There are some agent programming languages as well, such as 3APL [59], AgentSpeak [101] and AGENT-0 [113]. To specify the design of agent-based systems, there are design languages, such as concurrent MetateM [47] and DESIRE [15]. Agent logics, such as the BDI-logics of Cohen and Levesque [21] and Rao and Georgeff [102], could be used to specify requirements on agent systems and to verify whether a finished system indeed meets the requirements, if there is a well-defined relation between the logic used and the design language, programming language or architecture used. Unfortunately, this relation is often impossible to find, because of the lack of proper formal semantics for agent architectures and programming languages or because the semantics of the architecture or programming or design language is not covered by the semantics of the logical languages. In the chapters following this chapter, we will define programming languages with a properly defined formal operational semantics, in an attempt to address the poor formal basis of many agent models.

In this chapter, we focus on verification of agent systems. We don't choose a particular architecture, programming language or design language the agent system is built in, but offer a very general approach. Even if there is a well-defined relation between the specification language for requirements and the language the system is specified in, verification of agent systems is not an easy task. As agents may operate in a world that is constantly changing, and agent systems can consist of a number of interacting but independent agents, expressing behavioural requirements may lead to complex formulae. Nevertheless, verification is important, because it is the only way to guarantee that demands made on aspects of the system behaviour are satisfied. The high degree of complexity of agent system behaviour is as much the reason as the problem here: by simply checking the code of the agent system or by testing, establishing that the system shows proper behaviour is intractable. Proper functioning is often crucial, because agent systems are increasingly employed in circumstances where mistakes have important consequences, for example in Electronic Commerce. But in practice, verification of agent systems is neglected, because it is intricate.

So, means are needed to make verification of agent systems manageable. Developers of agent systems should be enabled to verify the system they are building, assisted by tools, even if they are not specialists in formal theory. Properties and proofs have to be intuitively clear to the verifier and even, at least to some degree, to the stakeholder(s) of the system, as verification results are part of the design rationale of the system. Also, time complexity of the verification process has to be controlled. This chapter discusses some principles that contribute to the support of verification of agent systems. These principles can be used for all agent systems, but here, they are applied in the context of a single agent that performs actions in a dynamic environment.

In [70] a compositional verification method was introduced; see also [40] for its relation to temporal multi-epistemic logic. We extend this method in this chapter, and we briefly describe it here. A multi-agent system contains several agents, which interact in a certain environment, the world. Desired system behaviour is formalised in system requirements. In compositional verification, proving that a detailed design of an agent system properly respects the behavioural requirements means finding properties of the agents in the system and properties of the world, from which the system properties can be proved. If the agents are built from components, then the properties that have to hold for the agents should be provable from properties of the sub-components of the agent. This way, properties are refined across different process abstraction levels. For each of the agents as well as for the environment of the system (the world), a specific set of the refined properties is imposed to ensure that the

combined system satisfies the overall requirements.

The use of compositionality in verification has the following advantages:

- \* *Reuse* of verification results is supported. Verified system components can be put into another agent system, and as long as the conditions on the environment of the component still hold, the properties proved for this component stay valid. Also, if there are proofs for top-level system properties from properties of the agents and the world, it doesn't matter in which way the agents are composed of sub-components, as long as the properties of the agents can be proved. The proof of the top-level properties is independent of details of the composition of the agents.
- \* The verification process becomes *less complex*, as proofs are more local, considering only one process abstraction level at a time. So, if an agent system contains agents which have sub-sub-sub-components, properties of these sub-sub-sub-components never occur in the proof of top-level system properties. Only when verifying sub-sub-components, properties of sub-sub-sub-components are used.

In [14] it was shown how this method can be applied to prove properties of a system of negotiating agents.

However, the principle of compositionality does not solve all problems. Even if compositionality is exploited, for nontrivial examples verification is a tedious process. Also, properties and proofs can still be very complex to read and to explain. To manage the complexity of the proofs, and to make their structure more transparent, additional structuring means and reuse facilities are necessary. This chapter contributes two manners to support proof structuring and reuse.

On the one hand a notion of *language abstraction* is introduced that facilitates structuring of properties and proofs in a formal manner. To this end, the language to describe properties of agent systems is extended with new, more abstract, constructs. Parts of formulae can be given an intuitively enlightening name. This leads to a more informal look and feel for properties and proofs, without losing any formal rigour. The abstracted notions form a higher-level language to describe system behaviour. The terminology of this language abstracts away from details of the system design, and is closer to the way human verifiers conceptualise system behaviour. There are a number of benefits:

- \* Properties and proofs are more readable and easier to understand.
- \* Coming up with properties and proofs becomes easier, as the words chosen for the abstracted formulae guide and focus the cognitive verification

process of the verification engineer, providing clean-cut central concepts.

- \* Verification becomes explainable, as part of the design rationale documentation of a system.

On the other hand, common characteristics of agent systems can be exploited to support *reuse*. A range of agent concepts is associated with the paradigm of agents. For example, most agents receive observations and communicated information from their environment and perform actions to manipulate their environment. For this to yield desired results, proper coordination with the environment is essential. Properties regarding this apply to many agent systems and thus are highly reusable. Support of reuse requires that a library of predefined templates of properties and proofs is available. By identifying generic elements in the structure of proofs and properties, *reusable systems of properties and proofs* can be constructed. To illustrate this, this chapter proposes a system of coordination properties for applications of agents acting in dynamic environments. The properties and proofs of this system are an example of the contents of the verification library. Some advantages of reuse are:

- \* Verification becomes faster. Often, the verification engineer only has to look up suitable properties and proofs from the verification library and customise these by instantiation.
- \* Verification becomes easier. The contents of the library are usually phrased using language abstraction, so properties and proofs are more intuitively clear, making them more easy to use.

In the following section, the generic system consisting of an agent acting in a dynamic environment is sketched. For this application class, we will prove action successfulness, which means that all actions the agent performs yield the expected effects. To achieve this, a system of coordination properties is given in Section 3.4, as well as part of a proof that the actions of the agent succeed. But first, in Section 3.3 temporal models of agent systems descriptions are presented. This section also presents the two languages to describe system behaviour, the detailed language and the abstract language, and the connection between them. In Section 3.5, the language abstraction mechanism is applied; abstract predicates are introduced for parts of properties, yielding an abstract language. Section 3.6 contains basic properties of the agent and the world. As these properties describe general features of the agent and the world and the interaction between them, they can be used in proofs of many different properties. Here, these properties contribute to the proofs of two of the properties from the

system of coordination properties, which we assumed to hold in the proof of action successfulness in Section 3.4. These proofs appear in Section 3.7 and 3.8. In these sections, we also present additional properties which we need for these specific proofs. Section 3.9 shows how the generic properties and proofs are used to verify a real system that is an instantiation of the generic architecture. Finally, Section 3.10 contains some conclusions, comparisons with other work and directions for future research.

## 3.2 The domain of agents in a dynamic world

In this section the characteristics of the *application class of a single agent in interaction with a dynamic environment* are briefly discussed. We will present a reusable system of properties for this class later on, describing proper *coordination* of the agent with its environment. As stated in the Introduction (Chapter 1) of this thesis, we use the term ‘coordination’ to refer to agent behaviour which takes into account the dynamics of the (physical or virtual) world and the behaviour of the other agents, both of which are partly unpredictable. Alongside the presentation of the generic class, we also show one specific instance of the class, which will serve as a running example throughout this chapter.

Agents that can perceive and act in a dynamic environment are quite common. An example is an agent for process control (e.g. in a chemical factory). For this class of single agent systems, an important property is *successfulness of actions*. This means that all actions the agent initiates in its environment yield their expected effects. In this chapter, we give a *generic methodical proof for successfulness of actions*. Because this property is to be proven for a class of systems, it is needed to *abstract* from domain-dependent details of systems and give a *generic architecture* that defines the class.

The specification of a generic architecture for a single agent in a dynamic

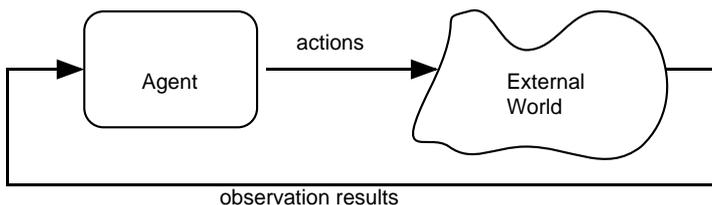


Figure 3.1: Agent and external world in interaction

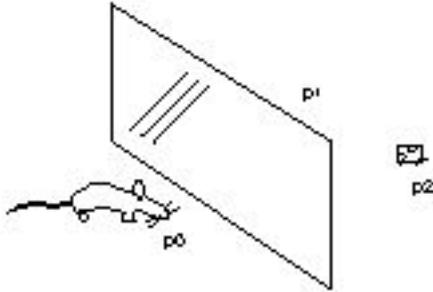


Figure 3.2: The mouse and the cheese

world depicted in Figure 3.1 consists of two components in interaction: an agent (Ag) and the external world (EW). Only a few aspects of the functioning of the system are specified by the architecture. Looking at the information flow in the system, actions generated by the agent are transferred from the agent's output interface to the external world, and observation results generated by the external world are transferred to the input interface of the agent. The connections between the agent and the world are called *links*. Based on the observation results the agent is to decide which actions are to be performed. The agent only receives information through observation results; it has no direct access to the world state. The role of the world in this system is to realise execution of all actions the agent decides to do, and to generate and execute events. The world also provides observations results and indications that actions have ended. Many concrete single agent systems are specialised instantiations of this generic architecture.

Throughout this chapter, we use a concrete system to illustrate the generic concepts. This concrete system is the *system of the artificial mouse*. This system models a mouse striving to be fed in a dynamic world. Using this system, we will show how to instantiate certain parameters of the generic class of single-agent systems in dynamic environments, in order to verify the concrete system. In Figure 3.2, we show the mouse in its environment. The agent in this system is the mouse. The mouse is located in a dynamic world, consisting of three positions (p0, p1 and p2, which are locations in the ground plane), a transparent screen, which can go up (and down also) and food (cheese) at p1 or p2. When the screen is down, it is in position p0. The world is dynamic in the sense that the screen, which can separate the mouse from the cheese, can go up and down. Initially, the mouse is at p0 (p0 can be the position of the mouse

as well as of the screen, without the mouse being squashed by the screen) and the screen is down. The mouse can perform two kinds of actions, eating and moving.

We will show the very simple implementation of the agent in the mouse system. The mouse is not very smart; as a decision model, it only has a knowledge base with two simple rules:

```

if    observation_result(at_position(food, P) ∧
                          ¬at_position(mouse, P) ∧
                          ¬at_position(screen, p0) )
then  to_be_performed(goto(P))

if    observation_result(at_position(mouse, P) ∧
                          at_position(food, P) )
then  to_be_performed(eat)

```

These rules are fairly obvious, except perhaps for the representation of variables. In the knowledge base,  $P$  is a variable, which can be instantiated with one of the three positions ( $p_0$ ,  $p_1$  and  $p_2$ ). When the agent obtains observation results of certain objects being in certain positions,  $P$  is instantiated with an actual position.

The first of these rules states that the mouse will move to a position where it sees food, when it is not itself at this position and the screen is up. The second rule means that the agent decides to eat as soon as it observes itself to be in the same place as the food. So, the mouse simply acts as soon as this seems to make sense. The mouse doesn't have any memory or complex goals; it just decides to do actions based on observations of occasions for the actions. We call agents that reason like this *reactive*; to be precise, reactive agents are agents that decide to do actions immediately when they observe occasions for these actions, without taking into account other circumstances, such as other actions that are still in execution or the chance an event will occur.

Though the dynamics of the world are rather limited, and the reasoning of the mouse is very simple, we experienced in earlier research that proving action successfulness for the mouse from scratch is extremely complicated and time-consuming.

We return to the generic class. Like in the previous chapter, we assume that the internal representation language the generic system employs is an order-sorted predicate logic (see Section 2.4). The formal ontologies of the system are very similar to the ontologies defined in Section 2.4.1. The names are

different and there is no formalisation of communication in this chapter, but there are no conceptual differences. The generic architecture contains formal ontologies for representing world information, observations, decisions to do actions and indications that actions have ended. The ontology for factual world information is called world info. As we are dealing with a generic architecture and the contents of this ontology is application-specific, we can't make general statements about this ontology, except that it is used to formalise facts about the state of the world. For example, in case of the concrete artificial mouse system, `at_position(food, p1)` is a formula based on world info. In the mouse system, `at_position` is the only relation in the ontology world info. This relation is binary; the first argument is an object (food, screen or mouse) and the second is a position (`p0`, `p1` or `p2`). The ontology world info was called EWOnt in the previous chapter. Just like previously, other ontologies are on a meta-level with respect to world info; expressions based on these higher level ontologies make statements *about* expressions from world info.

The *input interface* of the agent is defined by the formal ontologies observation results and actions finished. The ontology observation results is based on the sort containing all reified formulae based on world info and the relation `observation_result` on this sort. Formulae that can be expressed using this ontology are, for example, `observation_result(at_position(mouse, p0))` or `observation_result(at_position(food, p1)  $\wedge$   $\neg$ at_position(mouse, p1))`. The ontology actions finished is based on the sort ACTION (which is application-specific) and the relation `ended` on this sort. A formula based on this ontology is for example `ended(A)`, which means that the execution of action A has ended. We need `ended-atoms` to demarcate action executions; an action execution of action A will be the time period between a `to_be_performed(A)`-atom and a matching `ended(A)`-atom.

The *output interface* of the agent is defined by the formal ontology actions to be performed based on the sort ACTION and the unary relation `to_be_performed`. For example, the statement `to_be_performed(goto(p1))` can be expressed using this ontology. In the mouse system, ACTION contains four actions, namely `eat`, `goto(p0)`, `goto(p1)` and `goto(p2)`.

In contrast to the previous chapter, here we do focus on the interfaces of the world. For the external world the input and output interfaces are the opposite of the agent's interfaces, as the input of the world receives actions to be performed and the output of the world provides observation results and indications of the end of action executions to the agent. Additionally, the ontology world info is part of the output of the world. The agent only receives observation results, and never plain facts from world info. The reason world info is part of the definition of the output interface of the world is that this allows us to phrase properties

about action effects without having to look inside the world. We define the *agent input ontology* by  $\text{AgInOnt} = \text{observation results} \cup \text{actions finished}$  and the *agent output ontology* by  $\text{AgOutOnt} = \text{actions to be performed}$ . Also, we define the *agent ontology* by  $\text{AgOnt} = \text{AgInOnt} \cup \text{AgOutOnt}$ , the *world ontology* by  $\text{EWOnt} = \text{world info} \cup \text{actions to be performed} \cup \text{actions finished} \cup \text{observation results}$ , and the *overall ontology* by  $\text{OvOnt} = \text{AgOnt} \cup \text{EWOnt}$ .

Realistic characteristics of the agent systems in the class described above are:

- \* perceptions take time
- \* the generation of actions (reasoning, decision making) takes time
- \* execution of actions in the world takes time
- \* unexpected events can occur in the environment

Proving successfulness of actions under these circumstances is intricate because an action can only succeed when its execution is not disturbed too much. If two executions of actions overlap or events happen during executions, actions could fail. Also, while an agent is observing and reasoning, the situation in the world might change, such that when the agent has decided to do an action, this action is no longer successfully executable. We will propose a system of coordination properties that takes all these influences into account.

In the literature about action and change, varying attitudes towards these disturbances can be found. In one part of the literature (e.g., standard situation calculus, as described in [90, 104, 106]), these disturbances are excluded in a global manner, e.g., action generation and execution have no duration at all and no events occur at all. The problem with these global assumptions is that they violate the characteristics of most of the application domains. Other literature attaches default effects to actions, thus incorporating non-monotonic logic in action execution (e.g., [86] and [32]). Some literature takes into account duration of action execution (e.g., [109]). We didn't find any literature that also takes into account the duration of reasoning and decision processes in action generation. Another lack in the literature is that authors don't try to verify implemented systems; they only state theories regarding actions, without relating them to practical system design or software specification.

The set of properties we use to prove action successfulness contains properties of the *agent* and of the *world*, from which we will prove action successfulness. Thus, *we prove properties of the system, assuming properties of the agent and of the world*. When verifying an instance of the generic architecture, the agent and world properties which form the assumptions of the proofs in this

chapter have to hold for the particular agent and world in this specific system, and guaranteeing this will take additional verification effort. The benefit of this modular verification approach is that it enables reuse of verification results, and it allows us to abstract from details of the internal agent and of the internal dynamics of the world.

Part of the proof (and the properties involved) only applies to a subclass of the generic architecture, namely to *reactive* single agents acting in a dynamic environment. We have to make this restriction because proving the properties for the whole class of systems leads to restrictions of the world dynamics which are often too strong. Later on, in Section 3.6.2, we will elaborate on this issue.

Preliminary work on the analysis of this domain was reported in [72]. In continuation of this work we were led to the claims that:

- \* To prove action successfulness for realistic applications, properties are required that do not (completely) fix the dynamics of the interaction between agent and environment, but still impose enough structure on the process to prove that the agent is effective in its behaviour.
- \* These properties are explicitly related to practical system designs in a formal manner.
- \* Such properties exist: in this chapter we introduce a system of coordination properties that fulfils the above requirements.

These claims will be supported by the next sections.

## 3.3 Temporal models and temporal languages

For phrasing properties, a language is needed. Behaviour is described by properties of the execution traces of the system. In this section, we introduce the language used for this. Also, this section introduces the language abstraction formalism.

### 3.3.1 Basic concepts

By adding a formalisation of time to the language internally used in the generic system (see the previous section), we obtain a formal language for phrasing behavioural properties. With this language, properties of traces are described in a direct manner, meaning that there is a clear relation between the formal temporal language and the trace semantics of a system specification. We define the logical languages in the same way as we did in Chapter 2.

The *state language*  $SL(D)$  of a system component  $D$  is the (order-sorted) predicate logic language based on the interface ontologies of  $D$ . The state language based on a single ontology  $Ont$  is denoted by  $SL(Ont)$ . The formulae of these languages are called *state formulae*. The overall state language is  $SL$ ; this is a shorthand for  $SL(OvOnt)$ . An *information state*  $M$  of a component  $D$  is an assignment of truth values  $\{true, false, unknown\}$  to the set of ground atoms in  $SL(D)$ . At each moment in time during execution of an agent system including component  $D$ , there is one information state that fixes the truth values of the ground atoms that play a role in the interfaces of  $D$ . The set of all possible information states of  $D$  is denoted by  $IS(D)$ .

We assume the time frames are *linear with initial time point 0*. A time frame may be discrete or dense. The approach introduced here works for dense linear time models as well as for discrete linear time models, because finite variability provides means to discretize dense time (see Section 2.4.3). In the rest of this chapter, we assume finite variability holds for the traces studied. A *trace*  $\mathcal{M}$  of a component  $D$  over a time frame  $T$  is a sequence of information states  $(M^t)_{t \in T}$  in  $IS(D)$ . The semantics of a system specification is a set of traces, formalising all executions possible. The set of all traces of  $D$  that can result from executing  $D$  is named  $Traces(D)$ ; this set constitutes the semantics of  $D$ . Given a trace  $\mathcal{M}$  of component  $D$ , the information state of the input interface of component  $C$  at time point  $t$  is denoted by  $state(\mathcal{M}, t, input(C))$  where  $C$  is either  $D$  or a component within  $D$ . Analogously,  $state(\mathcal{M}, t, output(C))$  denotes the information state of the output interface of component  $C$  at time point  $t$ . To refer to the overall information state of the system (composed of the agent and the world) at a certain moment, we use  $state(\mathcal{M}, t)$ .

The information states (e.g. from traces) can be related to formulae via the satisfaction relation  $\models$ . If  $\varphi$  is a state formula from  $SL(C)$  expressed in the input ontology of component  $C$ , then

$$state(\mathcal{M}, t, input(C)) \models \varphi$$

denotes that  $\varphi$  is *true* in this state at time point  $t \in T$ .

These statements can be compared to *holds*-statements in situation calculus, described in [90, 106]. A difference, however, apart from notational differences, is that we refer to a trace and time point, and that we explicitly focus on a specific part of the system. Based on these statements, which only use predicate symbol  $\models$ , behavioural properties can be formulated in a formal manner in a sorted predicate logic with sorts  $T$  for time points,  $Traces(D)$  for traces of component  $D$  and  $SL(D)$  for state formulae. The usual logical connectives such as  $\neg, \wedge, \Rightarrow, \forall, \exists$  are employed to construct formulae, as well as  $<$  and  $=$  (to

compare moments in time). Note that arbitrary combinations of quantification over time are allowed in this language. The language defined in this manner is denoted by TL(D) (*Temporal Language* of D). The union of the temporal languages of the two system components (agent and world) is TL. An example of a formula of TL, in which S refers to the whole system, is:

$$\begin{aligned} \forall \mathcal{M} \in \text{Traces}(\mathbf{S}) : \\ \forall t_1 : \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models \text{to\_be\_performed}(\mathbf{A}) \quad \Rightarrow \\ \exists t_2 > t_1 : \text{state}(\mathcal{M}, t_2, \text{output}(\text{Ag})) \models \text{to\_be\_performed}(\mathbf{B}) \end{aligned}$$

This expresses that every decision of Ag to do action A is always followed by a later decision to do B.

In this chapter, we verify behavioural properties through mathematical proofs of formulae of TL. Usually, these properties start with  $\forall \mathcal{M} \in \text{Traces}(\dots)$ , which means that they describe a property of all execution traces of a system component. In this chapter, we assume the semantics of a system specification, specified as an architecture or in a programming or design language, has been obtained as a set of traces as defined above, using a global time frame T for all parts of the system. For an alternative semantics based on local time frames per component, we refer to the thesis of Pascal van Eck [39]. We prove behavioural properties of the generic system of a single agent in interaction with a dynamic environment from behavioural properties of the agent and behavioural properties of the world. This way, we abstract from the internal specification languages used to build the agent and the world (there could even be two different languages), and just focus our attention on composing the behaviour of the two system components. When using the generic proofs we provide, it is necessary to prove the behavioural properties of the agent and of the world, in order to guarantee proper system behaviour. For this, the semantics of the internal specification language(s) is essential. But for proving properties of the system from properties of the agent and the world, the semantics of the underlying formalism(s) for internal specification is irrelevant.

The languages TL(D) are built around constructs that enable the verifier to express properties in a detailed manner, staying in direct relation to the semantics of the design specification of the system. For example, the state formulae are directly related to information states of system components. But the detailed nature of the language also has disadvantages; properties tend to get long and complex. The formalism of language abstraction, described in Section 3.3.2, alleviates this considerably.

As we have assumed finite variability for all traces studied, we also have that the change pinpoint principle holds for these traces (see Section 2.4.3).

The change pinpoint principle allows introducing a number of abbreviations. We first repeat two definitions from Chapter 2:

$$\oplus \text{state}(\mathcal{M}, t_1, \text{interface}) \models \varphi \equiv \text{state}(\mathcal{M}, t_1, \text{interface}) \models \varphi \wedge (t_1 \neq 0 \Rightarrow \exists t_2 < t_1 \forall t \in [t_2, t_1) : \text{state}(\mathcal{M}, t, \text{interface}) \not\models \varphi)$$

$$\oplus \text{state}(\mathcal{M}, t_1, \text{interface}) \not\models \varphi \equiv \text{state}(\mathcal{M}, t_1, \text{interface}) \not\models \varphi \wedge (t_1 \neq 0 \Rightarrow \exists t_2 < t_1 \forall t \in [t_2, t_1) : \text{state}(\mathcal{M}, t, \text{interface}) \models \varphi)$$

The  $\oplus$ -notation, pronounced as *just*, is used to denote a change to a certain information state.

Closely related is the new  $\otimes_{t_1, t_2} \oplus$ -notation, defined as follows:

$$\otimes_{t_1, t_2} \oplus \text{state}(\mathcal{M}, t_2, \text{interface}) \models \varphi \equiv \oplus \text{state}(\mathcal{M}, t_2, \text{interface}) \models \varphi \wedge \forall t \in \langle t_1, t_2 \rangle : \neg \oplus \text{state}(\mathcal{M}, t, \text{interface}) \models \varphi$$

$$\otimes_{t_1, t_2} \oplus \text{state}(\mathcal{M}, t_2, \text{interface}) \not\models \varphi \equiv \oplus \text{state}(\mathcal{M}, t_2, \text{interface}) \not\models \varphi \wedge \forall t \in \langle t_1, t_2 \rangle : \neg \oplus \text{state}(\mathcal{M}, t, \text{interface}) \not\models \varphi$$

This notation can be used to say that the information state has just changed in some way at  $t_2$ , for the first time since  $t_1$ . Sometimes it can be useful to also have the dual operator; this states that the information state has just changed at  $t_2$  and that this is the most recent change like that prior to  $t_1$ . For this, we use the notation  $-\otimes_{t_1, t_2} \oplus$ , defined as:

$$-\otimes_{t_1, t_2} \oplus \text{state}(\mathcal{M}, t_2, \text{interface}) \models \varphi \equiv \oplus \text{state}(\mathcal{M}, t_2, \text{interface}) \models \varphi \wedge \forall t \in \langle t_2, t_1 \rangle : \neg \oplus \text{state}(\mathcal{M}, t, \text{interface}) \models \varphi$$

$$-\otimes_{t_1, t_2} \oplus \text{state}(\mathcal{M}, t, \text{interface}) \not\models \varphi \equiv \oplus \text{state}(\mathcal{M}, t, \text{interface}) \not\models \varphi \wedge \forall t \in \langle t_2, t_1 \rangle : \neg \oplus \text{state}(\mathcal{M}, t, \text{interface}) \not\models \varphi$$

### 3.3.2 The language abstraction formalism

Experience in nontrivial verification examples has taught us that the temporal expressions needed in proofs can become quite complex and unreadable. Also, details of the formalisation blur the generic agent concepts in properties. As a remedy, new language elements are added as abbreviations of complex temporal formulae. These new language elements are defined within a language AL(D) (meaning *Abstract Language* of component D) with generic sorts T for time points, Traces(D) for traces and SL(D) for state formulae; additional sorts are allowed, for example to abbreviate state formulae. The union of AL(Ag) and

AL(EW), the abstract languages of the two system components of the generic architecture, is called AL. As a simple example, for the property that there is an action execution starting in the world at  $t$  a new predicate `ActionExStarts` can be introduced. Then the property can be expressed in the abstract language:

$$\text{ActionExStarts}(A, t, \text{EW}, \mathcal{M})$$

which is defined as:

$$\oplus \text{state}(\mathcal{M}, t, \text{input}(\text{EW})) \models \text{to\_be\_performed}(A)$$

We use the convention to denote new, abstract language elements using this font. `ActionExStarts` is very similar to a “normal” first-order predicate, except that usually in syntactical formulae from predicate logic, no reference is made to the semantical models ( $\mathcal{M}$  in our language).

Semantics of these new language elements is defined as the semantics of the detailed formulae they abstract from. In logic the notion of *interpretation mapping* has been introduced to describe the interpretation of one logical language in another logical language, for example geometry in algebra (cf. [63], Chapter 5). The languages AL(D) and TL(D) can be related to each other by a fixed interpretation mapping from the formulae in AL(D) onto formulae in TL(D). In Section 3.5, we will define elements from AL as shorter forms of TL-formulae.

The languages AL(D) abstract from details of the design of component D and enable the verifier to concentrate on higher level (agent) concepts. Each abstract formula has the same semantics as the related detailed formula, such that the relation to design specification details isn’t lost. Proofs can be expressed either at the detailed or at the abstract level, and the results can be translated to the other level. Because formulae in the abstract level logic can be kept much simpler than in the detailed level logic, the proof relations expressed on that level are much more transparent.

### 3.4 Proving successfulness of actions

In Section 3.4.1, we give an informal introduction to the *system of coordination properties*. This set of properties is generic in the sense that it is usable for all agent systems that match the generic architecture from Section 3.2. The properties in it are called *coordination* properties, because in case the agent and the world behave accordingly, the actions of the agent are coordinated with the occurrences in the world in such a way that all actions succeed. The system itself appears in Section 3.4.2. To demonstrate the power of language abstraction,

we phrase the properties of the system of coordination properties in AL, *before* we define the language constructs of AL in terms of the lower level language TL (which we do in Section 3.5). In Section 3.4.3, we give part of the proof of action successfulness (the topmost part), assuming a number of the coordination properties. The proof is quite simple and doesn't require looking into lower-level details at all. Later on, in Section 3.7 and 3.8, we will prove two of the coordination properties which we assume to hold in Section 3.4. These proofs do descend to the detailed level of TL, which is why we present the proofs and the properties needed solely for these proofs in separate sections, instead of in this section.

### 3.4.1 Approaching the problem of proving successfulness of actions

For the application class described in Section 3.2, the aim is to prove that under the specified assumptions all actions executed in the agent system are successful, that is, yield all of their effects. To arrive at a reusable and intuitively pleasing proof, it was necessary and illuminating to separate the different aspects into a number of properties. These will constitute the system of coordination properties. Here, we informally explore the properties needed.

An action succeeds when its execution renders the appropriate effects. These effects have to happen during the execution of the action to be recognisable as effects of that particular action. Recall that actions in our view take time. Just like the start of an execution of A is indicated by a `to_be_performed(A)`-atom, the end is indicated by an `ended(A)`-atom. Action effects take place between the occurrence of these two atoms. It is essential that effects of an action take place *during* the action execution, as we have no other manner to establish the causal connection between the action and the effects.

Note that when an action is performed more than once, the same atoms indicate start and end of the action. We don't use specific action instances for specific occurrences of actions. This more realistic assumption imposes stronger demands on the properties, as the properties must provide a way to identify specific action instances.

We will formalise action successfulness in `COORD1`. `COORD1` is the first property in the system of coordination properties. All other properties we introduce will serve to prove `COORD1`. This property is a *system property*, which means that it can't be proven from correct behaviour of the agent only or from correct behaviour of the world only. `COORD1` *describes that the interaction between the agent and the world is such that action executions succeed.*

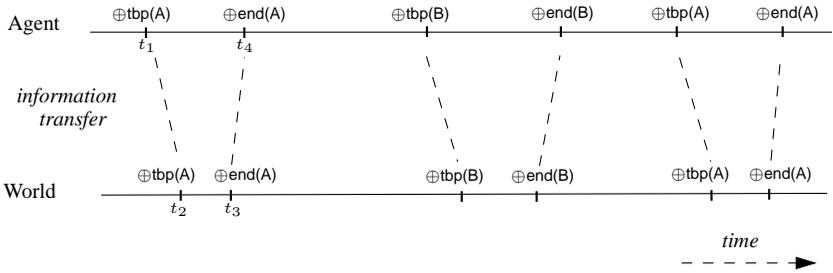


Figure 3.3: No overlapping of executions

Action executions can fail because of three reasons. The first reason is overlapping of action executions in time. If two action executions overlap, then the actions can interfere, leading to failure or unwanted effects. So, to guarantee success, action executions should not overlap. *Between the begin and the end of an action, no other actions should begin or end.* Property COORD0 formalises this. This property is also a system property. COORD0 states that the agent and the world interact in such a way that action executions don't overlap. This property is proved from other properties of the agent and the world using induction; we give the rather involved proof in Section 3.7. The properties needed for this proof constitute a sub-system of the system of coordination properties, and are named ORDER properties (as they describe the order in which things happen; see Section 3.7.1). Figure 3.3 illustrates COORD0. In figures, and sometimes also in the text, we will abbreviate *to\_be\_performed* to *tbp* and *ended* to *end*.

We will use figures like this frequently. In the figure, we show a trace of the agent (upper time line) and a trace of the world (lower time line). A  $\oplus\text{tbp}$ -atom or  $\oplus\text{end}$ -atom at a time point in a trace signifies that this atom has just become *true* in an interface of the agent or the world; the time point is often mentioned under the atom (see the leftmost execution of A in the figure).

COORD0 serves a double purpose; we need it in order to prove action successfulness (COORD1) and it enables us to *identify and distinguish specific action executions*. When *tbp*-atoms and *end*-atoms alternate in the depicted way, it is very easy to form pairs of *tbp*'s and *end*'s that demarcate a specific action instance execution: you simply take the first *end*(A) following the *tbp*(A). As is clear from Figure 3.3, it is easy to distinguish the first execution of A from the second execution of A. In the figure, there are action executions *in the world*, on the lower time line, as well as *in the agent*, on the upper time line; these terms will be used frequently. In Figure 3.4, we show a situation where action



Figure 3.4: Find the action execution . . .

executions overlap. It is clear that in this situation unambiguous identification of action executions is impossible.

COORD0 says that when one action is executing in the world, it is always the only one. It is also forbidden that two action executions entirely coincide. And because effects of an action are defined as expected outcomes that happen during execution of the action, no action execution can fail because of effects of other actions.

But action executions can also fail due to events. In our view, events are changes to the world state that aren't controlled by the agent. These can be due to the dynamics of the world itself (natural events). But changes due to the aftermath of a failed action also are events. For example, when you try to kill a fly by hitting it, this action has ended as soon as your hand is back in its original position. When the fly is still alive and kicking, you consider your action as a failure. But when the fly drops dead after another five minutes due to internal damage caused by your blow, we call this an event, even though strictly speaking your blow did kill it. But you didn't expect this to happen; normally the fly would have died during execution of the hitting action. Therefore, when the fly dies later on, this is seen as an event. And maybe the fly didn't even really die from the indirect consequences of your blow, but from a heart attack. This touches upon another point we should stress: when during execution of an action the expected consequence happens, this doesn't have to be caused by the action at all. In case of the fly: maybe it dies during your execution of the hitting action because of the fact that it was just old and nature called. But in this case, we see this as an effect of the action. This is reasonable, because there really is no other straightforward way of defining effects of actions. It is an old philosophical problem how to define causality, and we suspect these issues to be part of the problem. So, expected consequences happening in the world during an action execution are defined action effects, and all other changes are defined events.

So, in our view, each change to the world state either is the effect of a successful action or it is an event. We will formalise this in the property CHANGE, which we phrase in Section 3.6.3. In some other approaches, events designate all causes of state changes, including agent-initiated actions. This might be confusing, but is just a matter of definitions.

Property COORD3 is a property of the world, stating that *no events happen*

*during action executions*. This is quite a strong demand, as not every event might interfere with an action. The demand can be relieved by only forbidding interfering events, which is a minor extension. As COORD3 is a property of the world, we won't prove COORD3 in this chapter. We don't make any choices regarding the nature of the world; it could be the real external environment, or a simulated, implemented component. In the latter case, COORD3 should be proved from the semantics of the implementation of the world. But in this chapter, we restrict ourselves to the *interaction* between the agent and the world and to finding properties of the agent and of the world that will guarantee correct system behaviour. In this light, COORD3 is an *assumption* underlying the proof of COORD1; assuming that COORD3 holds (amongst other assumptions), we can prove COORD1.

A third reason for action failure is that the world can change prior to an action execution starting in the world. The agent decides to perform an action based on observation results it receives. If these observation results indicate there is an *occasion* for a certain action, which is a reason to do this action, then the agent will decide to do the action and generate a *tbp-atom* in its output interface. But between the moment the occasion arises in the world and the start of the execution of this action in the world, events or other action effects could occur, disrupting the applicability of the action. Property COORD5 states that *all actions are still applicable at the moment their execution starts in the world*. This is a system property, as both effects of actions (initiated by the agent) and events (taking place in the world) can be the cause of ruining the applicability of an action.

In order to prove COORD5, we need properties of the agent and of the world, just like we do for proving COORD0. We give these properties and the proof of COORD5 in Section 3.8. The properties are named *PROPER* properties (referring to the proper behaviour of agent and environment that doesn't affect applicability), and they constitute another sub-system of the system of coordination properties.

Both the sub-system for COORD0 and the sub-system for COORD5 are usable for a subclass of the generic architecture described in Section 3.2. In Sections 3.7 and 3.8, where we will go into the details of the proofs of these system properties, it becomes clear that proving COORD0 and COORD5 is not well possible without discerning different types of agents in the generic architecture described in Section 3.2. In this chapter, we will then focus on *reactive* agents, which are agents that decide to do actions immediately when they observe occasions for the actions, without taking into account other circumstances.

In Section 3.4.3, we show that absence of the three causes for failure laid down in the COORD properties is sufficient to prove successfulness of actions.

### 3.4.2 The system of coordination properties

One of the main objectives of this chapter is to establish a generic set of properties that enables the verifier to prove that all actions executed will succeed. We want to give a clear separation of all aspects involved, as described in the previous subsection, and to formalise demands on world and agent behaviour that guarantee overall action success. To obtain these goals, a system of coordination properties is devised. The properties are phrased in the abstract language AL, although the interpretation of this language in the detailed language is presented later on, to show the intuitive power of the language abstraction formalism.

The system is structured in the following way:

- \* COORD1 is the target, topmost property, formalising action successfulness and proved from all other properties;
- \* COORD0 is the foundational property, parts of which are frequently used as condition in other coordination properties;
- \* COORD2 and COORD5 are intermediate properties;
- \* COORD3 and COORD4 are demands on the world.

Actually, this is only the topmost part of the system of coordination properties; the other part is formed by the properties needed to prove COORD0 (Section 3.7) and COORD5 (Section 3.8).

COORD0 is the foundation of the system of coordination properties. It enables the verifier to identify action executions, by formalising Figure 3.3. The property states that action executions don't overlap, not in the world and neither in the agent. The agent- and the world-part will be part of the conditions of many properties to come, to enable identification of action executions. This is the abstract formula:

$$\begin{aligned} \text{COORD0 :} \\ \forall \mathcal{M} \in \text{Traces}(\mathcal{S}) : \\ \quad \text{NoOverlappingInWorld}(\mathcal{M}) \quad \wedge \\ \quad \text{NoOverlappingInAgent}(\mathcal{M}) \end{aligned}$$

The abstract predicates `NoOverlappingInWorld` and `NoOverlappingInAgent` will be defined in Section 3.5.

COORD1 formalises action successfulness, stating that all actions executed in the system are applicable and yield all expected effects. Informally:

‘When an action execution in the world begins at  $t_1$  and ends at  $t_2$ ,  
then  
the action is applicable at  $t_1$  in the world  
and  
all expected effects of the action in that world situation will be  
realised during the execution.’

This is the abstract formalisation:

$$\begin{aligned}
&\text{COORD1 :} \\
&\forall \mathcal{M} \in \text{Traces}(S) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
&\quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \Rightarrow \\
&\quad \text{Appl}(A, t_1, \mathcal{M}) \quad \wedge \\
&\quad \text{ExpEffectsHappen}(A, t_1, t_2, \mathcal{M})
\end{aligned}$$

It is essential that all action executions are applicable at the moment they start in the world. When actions are applicable at the moment the execution starts, the expected effects of the action are the desired effects. When an action is not applicable, there might be no effects at all, or unwanted ones.

COORD2 states that all action executions started at times that the action is applicable will be successful. Informally:

‘When an action execution in the world begins at  $t_1$  and ends at  $t_2$ ,  
and  
when the action is applicable at  $t_1$  in the world  
then  
all expected effects of the action in that world situation will be  
realised during the execution.’

And this is its abstract formalisation:

$$\begin{aligned}
&\text{COORD2 :} \\
&\forall \mathcal{M} \in \text{Traces}(S) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
&\quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \wedge \\
&\quad \text{Appl}(A, t_1, \mathcal{M}) \quad \Rightarrow \\
&\quad \text{ExpEffectsHappen}(A, t_1, t_2, \mathcal{M})
\end{aligned}$$

COORD3 is a demand on the world that states that there are no events happening during action executions. Informally:

‘If there is an action execution in the world  
and  
action executions do not overlap  
then  
no events happen during the execution.’

And this is the formalisation in the abstract language:

$$\begin{array}{l}
\text{COORD3 :} \\
\forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
\quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \wedge \\
\quad \text{NoOverlappingInWorld}(\mathcal{M}) \qquad \qquad \qquad \Rightarrow \\
\quad \text{NoEventsDuring}([t_1, t_2], \mathcal{M})
\end{array}$$

We need the `NoOverlappingInWorld` condition in this property because action executions can’t be properly identified when action beginnings and endings can occur at random.

`COORD4` is a demand on the world that says that an action execution in the world will be successful when the action is applicable and there are no disturbances caused by overlapping executions or events. As we sketched in the previous subsection, it is reasonable that actions succeed in these circumstances. Informally:

‘If an action execution in the world begins at  $t_1$  and ends at  $t_2$   
and  
action executions do not overlap  
and  
no events happen during the execution  
and  
the action is applicable at  $t_1$   
then  
all effects of the action will be realised during the execution.’

This is the formalisation:

$$\begin{array}{l}
\text{COORD4 :} \\
\forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
\quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \qquad \qquad \qquad \wedge \\
\quad \text{NoOverlappingInWorld}(\mathcal{M}) \qquad \qquad \qquad \wedge \\
\quad \text{NoEventsDuring}([t_1, t_2], \mathcal{M}) \qquad \qquad \qquad \wedge \\
\quad \text{AppI}(A, t_1, \mathcal{M}) \qquad \qquad \qquad \qquad \qquad \qquad \Rightarrow \\
\quad \text{ExpEffectsHappen}(A, t_1, t_2, \mathcal{M})
\end{array}$$

COORD5 simply states that an action is applicable at the moment its execution starts. This is a necessary condition for success of this action. This is its formalisation:

$$\begin{aligned} \text{COORD5 :} \\ \forall \mathcal{M} \in \text{Traces}(\mathcal{S}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\ \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \Rightarrow \\ \text{App1}(A, t_1, \mathcal{M}) \end{aligned}$$

Because the abstraction formalism is exploited, these properties are relatively easy to read and understand, even without knowing the formal meaning of abstract terms, which we provide in Section 3.5. Technical details are hidden beneath intuitively clear notions. The clarity and brevity of the formulae make the verification process more manageable, as the abstract concepts yield a natural view of the system’s behaviour and prevent getting lost in symbolic clutter while constructing proofs. The system of coordination properties is applicable for many systems with a single agent that performs actions in a changing world. By simple instantiation of the system specific details, such as the set of actions, the conditions of applicability and the effects of these actions, the system can be customised, as we will sketch in Section 3.9.

### 3.4.3 The proof of action successfulness

The proof of COORD1 from the other properties of the system of coordination properties is extremely simple. In order to prove COORD1, it is possible to stay entirely within the abstract language AL; no abstractions need to be expanded into the detailed language. All that is needed is performing simple modus ponens and manipulation of universal quantifiers on a subset of the system of coordination properties.

For the proof, we assume that the world properties COORD3 and COORD4 hold. We won’t prove these properties in this chapter, as it is our aim to prove the system property action successfulness (COORD1) from properties of the agent and the world. For an instance of the generic architecture, with some specific (real or simulated) world, COORD3 and COORD4 will need to hold in order for the proof to be valid. We also assume that the system properties COORD0 and COORD5 are valid. We will prove these properties in this chapter from other agent and world properties, in Section 3.7 and Section 3.8, respectively. The proofs of these system properties are quite involved and detailed.

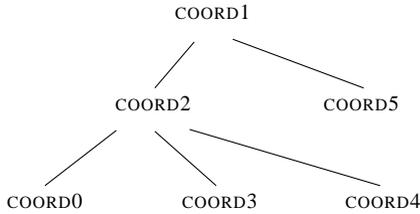


Figure 3.5: The proof tree

We show the proof tree of COORD1 in Figure 3.5. We will briefly explain the proof steps. To prove COORD2, we take any trace  $\mathcal{M}$  of  $\mathbf{S}$  (as COORD2 is a system property), an action  $A$  and two points in time,  $t_1$  and  $t_2 > t_1$ . We then suppose that  $\text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \wedge \text{App1}(A, t_1, \mathcal{M})$ . COORD0 gives us  $\text{NoOverlappingInWorld}(\mathcal{M})$ , and now we use COORD3 and modus ponens to conclude that  $\text{NoEventsDuring}([t_1, t_2], \mathcal{M})$ . Finally, we apply modus ponens to COORD4 and the other information we have, to conclude that  $\text{ExpEffectsHappen}(A, t_1, t_2, \mathcal{M})$ , with which we have proven COORD2.

Proving COORD1 is even simpler. We again take a trace  $\mathcal{M}$  of  $\mathbf{S}$ , an action  $A$  and two points in time,  $t_1$  and  $t_2 > t_1$ . We suppose that  $\text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M})$ . COORD5 then allows us to conclude  $\text{App1}(A, t_1, \mathcal{M})$ , and with this information, we can use COORD2 and modus ponens to arrive at  $\text{ExpEffectsHappen}(A, t_1, t_2, \mathcal{M})$ .

## 3.5 Relating abstract notions to detailed meaning

In this section, we will *define the language constructs* (predicates and terms) of the abstract language AL. We do this by relating them to formulae of the detailed temporal language TL. In Section 3.4, we already used AL notions to phrase coordination properties, and in later Sections (3.6, 3.7 and 3.8) we will present more properties written in the language AL. The abstract language enables expressing temporal properties of system behaviour using a vocabulary of clean-cut concepts. As we stated before, to distinguish elements of the abstract language, a different font is used for the constructs of the abstract language.

## Concerning action executions

The notion of an *action execution* is central to the system of coordination properties, so formalisation is desired. An action execution of A is a period in time starting with the appearance of some `to_be_performed(A)`-atom and ending with the appearance of the first subsequent `ended(A)`-atom. In the generic system, there are two perspectives on action executions. There is the action execution in the *world*, which happens between  $t_1$  and  $t_2$  when a `tbp`-atom appears in the input interface of the world at  $t_1$  and the first matching end-atom appears in the output interface of the world at  $t_2$ . And there is the action execution in the *agent*, which happens between  $t_1$  and  $t_2$  when a `tbp`-atom appears in the output interface of the agent at  $t_1$  and the first matching end-atom appears in the input interface of the agent at  $t_2$ . We need these two variants of action executions because we sometimes look at agent properties, and other times at environment properties.

The definitions of the abstract notions of action executions only yield the right intuitions when property `COORD0` holds, as explained in the previous section. If not, then it is not reasonable to take on the first matching end-atom as belonging to the `tbp`-atom, as it could be the end of the execution of another instance of the same action, which started at some earlier time point.

Besides action execution intervals, we also define notions for action executions starting or ending. First, we introduce new predicates and explain them in informal terms. Next, we give formal interpretations in terms of the detailed language.

Let  $A \in \text{ACTION}$  and  $t_1, t_2 > t_1$  be moments in time. Then, the abstract formula

$\text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M})$  denotes that there is an execution of A in the world starting at  $t_1$  and ending at  $t_2$ , and

$\text{ActionEx}(A, t_1, t_2, \text{Ag}, \mathcal{M})$  denotes that there is an execution of A in the agent starting at  $t_1$  and ending at  $t_2$ .

Interpretation in terms of the detailed language:

$\text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \equiv \oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{EW})) \models \text{to\_be\_performed}(A) \wedge \otimes_{t_1, t_2} \oplus \text{state}(\mathcal{M}, t_2, \text{output}(\text{EW})) \models \text{ended}(A)$

$\text{ActionEx}(A, t_1, t_2, \text{Ag}, \mathcal{M}) \equiv \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models \text{to\_be\_performed}(A) \wedge \otimes_{t_1, t_2} \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models \text{ended}(A)$

Let  $A \in \text{ACTION}$  and  $t_1$  be a moment in time. Then, the abstract formula  
 $\text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M})$  denotes that an execution of  $A$  in the world starts at  $t_1$ , and  
 $\text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M})$  denotes that an execution of  $A$  in the agent starts at  $t_1$ .

Interpretation in terms of the detailed language:

$$\begin{aligned} \text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M}) &\equiv \oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{EW})) \models \text{to\_be\_performed}(A) \\ \text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) &\equiv \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models \text{to\_be\_performed}(A) \end{aligned}$$

Let  $A \in \text{ACTION}$  and  $t_2$  be a moment in time. Then, the abstract formula  
 $\text{ActionExEnds}(A, t_2, \text{EW}, \mathcal{M})$  denotes that an execution of  $A$  in the world ends at  $t_2$ , and  
 $\text{ActionExEnds}(A, t_2, \text{Ag}, \mathcal{M})$  denotes that an execution of  $A$  in the agent ends at  $t_2$ .

Interpretation in terms of the detailed language:

$$\begin{aligned} \text{ActionExEnds}(A, t_2, \text{EW}, \mathcal{M}) &\equiv \oplus \text{state}(\mathcal{M}, t_2, \text{output}(\text{EW})) \models \text{ended}(A) \\ \text{ActionExEnds}(A, t_2, \text{Ag}, \mathcal{M}) &\equiv \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models \text{ended}(A) \end{aligned}$$

### Concerning first and consecutive actions

We sometimes need to identify the start of the first action in an interval in time as well as the beginnings of subsequent consecutive actions. We introduce two abstract notions for this, that both come in an agent variant and a world variant.

Let  $A \in \text{ACTION}$ ,  $int$  be an interval in time and  $t_1$  be a moment in time such that  $t_1 \in int$ . Then, the abstract formula

$\text{FirstActionBegins}(A, t_1, int, \text{EW}, \mathcal{M})$  denotes that  $A$  is the first action starting in the world during  $int$ , and it starts at time  $t_1$ , and

$\text{FirstActionBegins}(A, t_1, int, \text{Ag}, \mathcal{M})$  denotes that  $A$  is the first action in  $int$  the agent decides to do, and the execution in the agent begins at  $t_1$ .

Interpretation in terms of the detailed language:

$$\begin{aligned} \text{FirstActionBegins}(A, t_1, \text{int}, \text{EW}, \mathcal{M}) &\equiv \\ &\text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M}) \wedge \\ &\forall t \in [0, t_1) \cap \text{int} \forall B \in \text{ACTION} : \\ &\neg \text{ActionExStarts}(B, t, \text{EW}, \mathcal{M}) \end{aligned}$$

$$\begin{aligned} \text{FirstActionBegins}(A, t_1, \text{int}, \text{Ag}, \mathcal{M}) &\equiv \\ &\text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) \wedge \\ &\forall t \in [0, t_1) \cap \text{int} \forall B \in \text{ACTION} : \\ &\neg \text{ActionExStarts}(B, t, \text{Ag}, \mathcal{M}) \end{aligned}$$

Here,  $[0, t_1) \cap \text{int}$  is the sub-interval of  $\text{int}$  of all time points before  $t_1$ . As there is an action starting at  $t_1$ , no other actions may start in this sub-interval for this action to be the first one.

Let  $A, B \in \text{ACTION}$  and  $t_1, t_2 > t_1$  be moments in time. Then, the abstract formula

$\text{ConsecutiveActionsBegin}(A, B, t_1, t_2, \text{EW}, \mathcal{M})$   
denotes that A and B are consecutive actions in the world, starting at  $t_1$  and  $t_2$ , respectively, and

$\text{ConsecutiveActionsBegin}(A, B, t_1, t_2, \text{Ag}, \mathcal{M})$   
denotes that A and B are consecutive actions that the agent decides to do, starting at  $t_1$  and  $t_2$ , respectively.

Interpretation in terms of the detailed language:

$$\begin{aligned} \text{ConsecutiveActionsBegins}(A, B, t_1, t_2, \text{EW}, \mathcal{M}) &\equiv \\ &\text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M}) \wedge \\ &\text{ActionExStarts}(B, t_2, \text{EW}, \mathcal{M}) \wedge \\ &\forall t \in \langle t_1, t_2 \rangle \forall C \in \text{ACTION} : \\ &\neg \text{ActionExStarts}(C, t, \text{EW}, \mathcal{M}) \end{aligned}$$

$$\begin{aligned} \text{ConsecutiveActionsBegins}(A, B, t_1, t_2, \text{Ag}, \mathcal{M}) &\equiv \\ &\text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) \wedge \\ &\text{ActionExStarts}(B, t_2, \text{Ag}, \mathcal{M}) \wedge \\ &\forall t \in \langle t_1, t_2 \rangle \forall C \in \text{ACTION} : \\ &\neg \text{ActionExStarts}(C, t, \text{Ag}, \mathcal{M}) \end{aligned}$$

## Concerning applicability

Actions can only be successfully executed in certain world states. There must be nothing obstructing the execution of the action. For each action A, we assume there is a state formula  $\text{appl}(A) \in \text{SL}(\text{world info})$ , describing exactly the

world situations in which the action can be executed. It is not excluded that the effects of the action are already present in these world situations; in this case, execution of the action could leave the world state unaltered.

The formulae  $appl(A)$  are parameters of the generic system of properties, that have to be filled in when verifying an instantiation of the generic architecture. For example, for the artificial mouse (further details in Section 3.2 and 3.9), we use:

$$\begin{aligned}
 appl(\text{goto}(p2)) &\equiv (\text{at\_position}(\text{mouse}, p0) \wedge \neg \text{at\_position}(\text{screen}, p0)) \vee \\
 &\quad \text{at\_position}(\text{mouse}, p1) \\
 appl(\text{eat}) &\equiv (\text{at\_position}(\text{mouse}, p0) \wedge \text{at\_position}(\text{food}, p0)) \vee \\
 &\quad (\text{at\_position}(\text{mouse}, p1) \wedge \text{at\_position}(\text{food}, p1)) \vee \\
 &\quad (\text{at\_position}(\text{mouse}, p2) \wedge \text{at\_position}(\text{food}, p2))
 \end{aligned}$$

The choice of these two formulae is obvious when looking at Figure 3.2 (in Section 3.2). For example, if we look at the action  $\text{goto}(p2)$ , it is clear that the mouse can go to  $p2$  either if it is at the other side of the screen (at  $p0$ ) and the screen is up, or it is at  $p1$ , where nothing can obstruct its movements. Here, we chose not to include the possibility that the mouse already is at  $p2$  in the applicability of  $\text{goto}(p2)$ , though this might be an arbitrary choice.

Given these formulae that state in which circumstances each action is applicable, a temporal applicability notion can be defined straightforwardly:

Let  $A \in \text{ACTION}$  and  $t_1$  be a moment in time. Then, the abstract formula  $\text{App1}(A, t_1, \mathcal{M})$  denotes that action  $A$  is applicable in the world at  $t_1$ .

Interpretation in terms of the detailed language:

$$\text{App1}(A, t_1, \mathcal{M}) \equiv \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models appl(A)$$

### Concerning occasionality

The agents must have a reason in order to decide to do some action. Part of the reason must be the world situation; you only decide to do an action when the world situation is such that doing the action might be useful. If the world situation is such that the agent thinks it is a good idea to perform action  $A$ , we say that the world provides an *occasion* for  $A$ . For example, if you want to cross a busy road, and the traffic light turns green, this provides an occasion for you to cross the road.

Note that presence of occasion for an action is not the same as the action being applicable: an occasion is a state of the world that eventually (by

observation and deliberation) causes the agent to decide to do an action, and applicability means that the world state is such that an action can be fruitfully executed. In case the agent can obtain sufficient information about the state of the world and the knowledge of the agent is correct, the fact that there is an occasion for an action should imply that this action is also applicable. Otherwise, the agent would decide to perform an action that can't be executed, which doesn't make any sense. Note that when the agent has only partial information on the world, it might be the case that occasionality doesn't always imply applicability. In these circumstances, the agent must sometimes decide to do an action, without being sure that the action is successfully executable. But for proving action successfulness, we will rely on the property that occasionality implies applicability.

Just like applicability, occasionality is a system- and domain dependent notion. So, we assume that for each action  $A$  there is a state formula  $occ(A) \in \text{SL}(\text{world info})$ , describing exactly the world situations that form occasions for action  $A$ . For example, in the mouse system, we have:

$$\begin{aligned} occ(\text{goto}(P)) &\equiv \text{at\_position}(\text{food}, P) \wedge \neg \text{at\_position}(\text{mouse}, P) \wedge \\ &\quad \neg \text{at\_position}(\text{screen}, p0) \\ occ(\text{eat}) &\equiv \exists P : \text{at\_position}(\text{mouse}, P) \wedge \text{at\_position}(\text{food}, P) \end{aligned}$$

The formulae closely match the conditions of the rules of the knowledge base of the mouse, which we gave in Section 3.2.

Just like with applicability, we define an AL-notion for an occasion taking place in the external world. But this is not enough, as the agent has to *observe* the occasion before it decides to do anything. Just like acting, the process of observing something might take a while, and so we define notions for an occasion being observable in the output interface of the world and for an occasion being observed in the input interface of the agent.

Let  $A \in \text{ACTION}$  and  $t_1$  be a moment in time. Then, the abstract formula  $\text{Occ}(A, t_1, \mathcal{M})$  denotes that there is an occasion for action  $A$  in the world at  $t_1$ .

Interpretation in terms of the detailed language:

$$\text{Occ}(A, t_1, \mathcal{M}) \equiv \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models occ(A)$$

Let  $A \in \text{ACTION}$  and  $t_1$  be a moment in time. Then, the abstract formula  $\text{ObsOcc}(A, t_1, \text{EW}, \mathcal{M})$  denotes that an occasion for action  $A$  is observable in the world at  $t_1$ , and

$\text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M})$  denotes that the agent observes an occasion for action  $A$  at  $t_1$ .

Interpretation in terms of the detailed language:

$$\begin{aligned} \text{ObsOcc}(A, t_1, \text{EW}, \mathcal{M}) &\equiv \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models \\ &\quad \text{observation\_result}(\text{occ}(A)) \\ \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M}) &\equiv \text{state}(\mathcal{M}, t_1, \text{input}(\text{Ag})) \models \\ &\quad \text{observation\_result}(\text{occ}(A)) \end{aligned}$$

Returning to the example above,  $\text{ObsOcc}(\text{goto}(P), t, \text{Ag}, \mathcal{M})$  is a shorthand for

$$\begin{aligned} \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models &\quad \text{observation\_result}(\text{at\_position}(\text{food}, P) \wedge \\ &\quad \neg \text{at\_position}(\text{mouse}, P) \wedge \\ &\quad \neg \text{at\_position}(\text{screen}, p0)) \end{aligned}$$

and truth of this formula makes the mouse decide to go to  $P$ .

### Concerning situation specific effects and expected effects

When an action is executed in a certain world state, there (possibly) will be some effects. By an *effect* is meant a single ground literal from  $\text{SL}(\text{world info})$  that becomes *true* as a result of an action. For the sake of simplicity, we don't consider more involved effects, such as disjunctive effects. An action can have multiple effects, each happening at some moment during the execution of the action. The exact nature of the effects depends on the world situation; in this way, the effects are situation specific. For example, if an agent throws a ball forward when standing faced towards a lake, then a first effect of this action is that the ball is no longer in the actuators of the agent. Later on in the action execution, when the ball lands, another effect is that the ball gets wet. When this same action is executed in a different world situation, for example when the agent faces a glass office building, then the second effect of the action will be very different.

Before we formalise world situations, we explore the nature of the representations in component EW. In Section 3.3.1, we defined information states of components to be three-valued; each interface atom can be *true*, *false* or *unknown*. If an atom is *unknown* in an information state, this means that the component doesn't have information regarding this atom. For the information in the output interface of the world, we exclude the possibility of *unknown* atoms. The world state is represented using atoms from  $\text{SL}(\text{world info})$ . In information states of the world (that is, of component EW), ground atoms based on world info can be either *true* or *false*, and never *unknown*. Later on, we introduce a world property formalising this (2VAL, Section 3.6.3). The intuition for this choice is that there is no uncertainty regarding facts in the world (if we

don't consider quantum-mechanical phenomena). It either rains in New York at this very moment, or it doesn't. *You* probably don't know which of the two is the case, but it is clear that "It rains now in New York" either is *true* or *false*.

Now, a *world situation* is formalised as a partial assignment of definite truth values (*true* and *false*) to ground atoms of world info. We use a partial assignment because a world situation often concerns only part of the ground atoms forming the representation of the world state; the truth values of other atoms are not relevant. Each world situation can be refined to a number of complete assignments. A world situation can be represented as a conjunction of ground literals of world info. An example of a world situation in the mouse system is  $\text{at\_position}(\text{mouse}, p0) \wedge \neg \text{at\_position}(\text{screen}, p0)$ . A world situation  $W'$  is a refinement of a world situation  $W$ , written as  $W \leq W'$ , if their conjunction representations obey:  $W' \Rightarrow W$ .

The effects of actions are domain- and system dependent notions. For each system  $S$ , a relation *sit\_spec\_effect* is defined to represent the situation specific effects of actions. This relation is one of the parameters of the reusable set of properties, that allows instantiation to a specific system. Given three arguments (an action  $A$ , a world situation  $W$ , and  $l$ , a groundliteral of type world info), *sit\_spec\_effect*( $A, W, l$ ) means that literal  $l$  becoming *true* is an effect of the action  $A$  in the world situation  $W$ . This system-dependent relation must always satisfy the following demand:

For all  $A \in \text{ACTION}$ , all groundliterals  $l$  of type world info and all world situations  $W$  and  $W'$ :

$$W \leq W' \wedge \text{sit\_spec\_effect}(A, W, l) \Rightarrow \text{sit\_spec\_effect}(A, W', l).$$

This property is monotonicity of effects with respect to situations: when more facts are available concerning the state of the world, the set of predicted effects will be larger or the same.

The predicate *sit\_spec\_effect* described above is static; it doesn't have a trace ( $\mathcal{M}$ ) or moment in time as an argument. Using *sit\_spec\_effect*, we define a dynamic notion for effects that are expected as a consequence of performing an action in the world at a certain moment in time. When an execution of an action  $A$  starts at  $t_1$  in the world, the effects expected depend on the factual world situation at  $t_1$ . So, the following definition takes into account the output information state of  $EW$ , restricted to the factual information type world info.

Let  $A \in \text{ACTION}$ ,  $l \in \text{groundliterals}(\text{world info})$  and  $t_1$  be a moment in time. Then, the abstract formula

$\text{ExpEffect}(l, A, t_1, \mathcal{M})$

denotes that  $l$  is expected to happen as a result of executing  $A$  in the world at  $t_1$ .

This formula is defined within the abstract language by:

$\text{ExpEffect}(l, A, t_1, \mathcal{M}) \equiv$

$\text{sit\_spec\_effect}(A, \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW}))|_{\text{world info}}, l)$

The second argument of *sit\_spec\_effect* above is

$\text{state}(\mathcal{M}, t_1, \text{output}(\text{EW}))|_{\text{world info}}$ , which yields the truth values of all ground atoms in *world info* in the information state at  $t_1$  in trace  $\mathcal{M}$ . As the world is two-valued, this is a world situation, so *sit\_spec\_effect* can be applied to it.

We give an example from the mouse system. We focus on the action  $\text{goto}(p1)$ . We have:

$\text{sit\_spec\_effect}(\text{goto}(p1), \text{at\_position}(\text{mouse}, p0) \wedge \neg\text{at\_position}(\text{screen}, p0),$   
 $\text{at\_position}(\text{mouse}, p1))$  and  
 $\text{sit\_spec\_effect}(\text{goto}(p1), \text{at\_position}(\text{mouse}, p0) \wedge \neg\text{at\_position}(\text{screen}, p0),$   
 $\neg\text{at\_position}(\text{mouse}, p0))$

As moving only has two effects, the above clauses specify all effects of the mouse going to  $p1$  in the described world situation. We now look at a trace  $\mathcal{M}$ , such that  $\text{state}(\mathcal{M}, t_1, \text{output}(\text{EW}))|_{\text{world info}}$  is the world situation:

$\text{at\_position}(\text{mouse}, p0) \wedge \neg\text{at\_position}(\text{mouse}, p1) \wedge \neg\text{at\_position}(\text{mouse}, p2) \wedge$   
 $\neg\text{at\_position}(\text{food}, p0) \wedge \text{at\_position}(\text{food}, p1) \wedge \neg\text{at\_position}(\text{food}, p2) \wedge$   
 $\neg\text{at\_position}(\text{screen}, p0) \wedge \neg\text{at\_position}(\text{screen}, p1) \wedge \neg\text{at\_position}(\text{screen}, p2)$

We call this world situation  $W$ . We have that

$\text{at\_position}(\text{mouse}, p0) \wedge \neg\text{at\_position}(\text{screen}, p0) \leq W$ ,

and thus the situation specific effects for  $\text{at\_position}(\text{mouse}, p0) \wedge \neg\text{at\_position}(\text{screen}, p0)$  also happen in  $W$ . We then have

$\text{ExpEffect}(\text{at\_position}(\text{mouse}, p1), \text{goto}(p1), t_1, \mathcal{M})$  and  
 $\text{ExpEffect}(\neg\text{at\_position}(\text{mouse}, p0), \text{goto}(p1), t_1, \mathcal{M})$ .

## Concerning effects of actions and events

A literal is defined to be an *effect of an action* when the literal is an expected outcome that becomes *true* during execution of the action. To be precise, if an execution of  $A$  starts at  $t_1$  and ends at  $t_2$ , then all expected effects of  $A$  happening during  $\langle t_1, t_2 \rangle$  are defined to be effects of  $A$ . We exclude the starting point of the action execution interval, because at the very moment that an action starts to be executed in the world, there can't be any effects yet. Note that when a literal is an action effect, this doesn't mean that the literal becomes *true* as a result of the action, though this will be usually the case. But when during an action execution an event happens, which causes changes that are also expected effects of the action being executed, these changes will be seen as effects of the action (recall the example of the fly in Subsection 3.4.1). This choice is made because there is no means by which an external observer can distinguish changes caused by actions from changes caused by events. A literal is defined to be an *effect of an event* when it is not an effect of any action. Before we define effects of actions and events, we first introduce a notion for something happening in the world situation. Something happens when a literal of world info becomes *true*.

Let  $l \in \text{groundliterals}(\text{world info})$  and  $t$  be a moment in time. Then, the abstract formula

$\text{Happens}(l, t, \mathcal{M})$  denotes that at  $t$ ,  $l$  becomes *true* in the world.

Interpretation in terms of the detailed language:

$\text{Happens}(l, t, \mathcal{M}) \equiv \oplus \text{state}(\mathcal{M}, t, \text{output}(\text{EW})) \models l$

Let  $A \in \text{ACTION}$ ,  $l \in \text{groundliterals}(\text{world info})$  and  $t$  be a moment in time. Then, the abstract formula

$\text{ActionEff}(A, l, t, \mathcal{M})$  denotes that at  $t$ ,  $l$  happens as a result of executing  $A$ .

Interpretation in terms of the detailed language:

$\text{ActionEff}(A, l, t, \mathcal{M}) \equiv \exists t_1 < t \exists t_2 \geq t:$   
 $\text{Happens}(l, t, \mathcal{M}) \wedge$   
 $\text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \wedge$   
 $\text{ExpEffect}(l, A, t_1, \mathcal{M})$

Let  $l \in \text{groundliterals}(\text{world info})$  and  $t$  be a moment in time. Then, the abstract formula

$\text{EventEff}(l, t, \mathcal{M})$  denotes that at  $t$ ,  $l$  happens as a result of some event.

Interpretation:

$$\begin{aligned} \text{EventEff}(l, t, \mathcal{M}) \equiv & \quad \text{Happens}(l, t, \mathcal{M}) \wedge \\ & \neg \exists A \in \text{ACTION}: \\ & \quad \text{ActionEff}(A, l, t, \mathcal{M}) \end{aligned}$$

The next abstract formula is used to state that during an interval in time there are no effects of events.

Let  $int$  be an interval in time. Then, the abstract formula

$\text{NoEventsDuring}(int, \mathcal{M})$  denotes that there are no events taking place in the world during  $int$ .

Definition within the abstract language:

$$\begin{aligned} \text{NoEventsDuring}(int, \mathcal{M}) \equiv & \quad \forall l \in \text{groundliterals}(\text{world info}) \\ & \quad \forall t \in int : \\ & \quad \neg \text{EventEff}(l, t, \mathcal{M}) \end{aligned}$$

### Concerning successful actions

The following formula of the abstract language states that an execution of  $A$  is successful, meaning that all expected effects are achieved during the execution:

Let  $A \in \text{ACTION}$  and  $t_1, t_2 > t_1$  be moments in time. Then, the abstract formula

$\text{ExpEffectsHappen}(A, t_1, t_2, \mathcal{M})$  denotes that all expected effects of doing  $A$  between  $t_1$  and  $t_2$  are achieved.

Definition within the abstract language:

$$\begin{aligned} \text{ExpEffectsHappen}(A, t_1, t_2, \mathcal{M}) \equiv & \quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \wedge \\ & \quad \forall l \in \text{groundliterals}(\text{world info}): \\ & \quad (\text{ExpEffect}(l, A, t_1, \mathcal{M}) \Rightarrow \\ & \quad \exists t_3 \in \langle t_1, t_2 \rangle: \\ & \quad \quad \text{ActionEff}(A, l, t_3, \mathcal{M})) \end{aligned}$$

### Concerning overlapping executions

The first two abstract formulae state that a certain execution, in the world or the agent respectively, doesn't overlap with any other execution.

Let  $A \in \text{ACTION}$  and  $t_1, t_2 > t_1$  be moments in time. Then, the abstract formula

$\text{NoOverlapDuringExecuting}(A, t_1, t_2, \text{EW}, \mathcal{M})$

denotes that in the world, between the beginning of an execution of  $A$  at  $t_1$  and the end of an execution of  $A$  at  $t_2$ , no other executions begin or end.

Definition within the abstract language:

$$\begin{aligned} \text{NoOverlapDuringExecuting}(A, t_1, t_2, \text{EW}, \mathcal{M}) &\equiv \\ &\text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M}) \wedge \\ &\text{ActionExEnds}(A, t_2, \text{EW}, \mathcal{M}) \quad \Rightarrow \\ &(\forall B \neq A \in \text{ACTION} \forall t \in [t_1, t_2] : \\ &\quad \neg \text{ActionExStarts}(B, t, \text{EW}, \mathcal{M}) \wedge \\ &\quad \neg \text{ActionExEnds}(B, t, \text{EW}, \mathcal{M})) \quad \wedge \\ &(\forall t \in \langle t_1, t_2 \rangle : \neg \text{ActionExStarts}(A, t, \text{EW}, \mathcal{M})) \wedge \\ &(\forall t \in [t_1, t_2] : \neg \text{ActionExEnds}(A, t, \text{EW}, \mathcal{M})) \end{aligned}$$

Let  $A \in \text{ACTION}$  and  $t_1, t_2 > t_1$  be moments in time. Then, the abstract formula

$\text{NoOverlapDuringExecuting}(A, t_1, t_2, \text{Ag}, \mathcal{M})$

denotes that in the agent, between the decision to begin an execution of  $A$  at  $t_1$  and the end of an execution of  $A$  at  $t_2$ , no other executions begin or end.

Definition within the abstract language:

$$\begin{aligned} \text{NoOverlapDuringExecuting}(A, t_1, t_2, \text{Ag}, \mathcal{M}) &\equiv \\ &\text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) \wedge \\ &\text{ActionExEnds}(A, t_2, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\ &(\forall B \neq A \in \text{ACTION} \forall t \in [t_1, t_2] : \\ &\quad \neg \text{ActionExStarts}(B, t, \text{Ag}, \mathcal{M}) \wedge \\ &\quad \neg \text{ActionExEnds}(B, t, \text{Ag}, \mathcal{M})) \quad \wedge \\ &(\forall t \in \langle t_1, t_2 \rangle : \neg \text{ActionExStarts}(A, t, \text{Ag}, \mathcal{M})) \wedge \\ &(\forall t \in [t_1, t_2] : \neg \text{ActionExEnds}(A, t, \text{Ag}, \mathcal{M})) \end{aligned}$$

Next, four abstract formulae are introduced that apply to a number of executions, namely the executions during a certain interval in time. There are two world notions and two agent notions. The abbreviations state that all executions started during the interval will also end in it, without being overlapped by other executions and that all executions ended during the interval have started during the interval, without being overlapped.

Let  $int$  be an interval in time. Then, the abstract formula

$ExecutionsStartedEndWithoutOverlapping(int, EW, \mathcal{M})$

denotes that all actions executions started in the world during  $int$  end in the interval, and there is no overlapping of executions during this interval.

Definition within the abstract language:

$ExecutionsStartedEndWithoutOverlapping(int, EW, \mathcal{M}) \equiv$

$\forall A \in \text{ACTION} \forall t_1 \in int :$

$\text{ActionExStarts}(A, t_1, EW, \mathcal{M}) \Rightarrow$

$\exists t_2 > t_1 : t_2 \in int \wedge$

$\text{ActionExEnds}(A, t_2, EW, \mathcal{M}) \wedge$

$\text{NoOverlapDuringExecuting}(A, t_1, t_2, EW, \mathcal{M})$

Let  $int$  be an interval in time. Then, the abstract formula

$ExecutionsEndedStartWithoutOverlapping(int, EW, \mathcal{M})$

denotes that all actions executions ended in the world during  $int$  have started in the interval, and there is no overlapping of executions during this interval.

Definition within the abstract language:

$ExecutionsEndedStartWithoutOverlapping(int, EW, \mathcal{M}) \equiv$

$\forall A \in \text{ACTION} \forall t_2 \in int :$

$\text{ActionExEnds}(A, t_2, EW, \mathcal{M}) \Rightarrow$

$\exists t_1 < t_2 : t_1 \in int \wedge$

$\text{ActionExStarts}(A, t_1, EW, \mathcal{M}) \wedge$

$\text{NoOverlapDuringExecuting}(A, t_1, t_2, EW, \mathcal{M})$

Let  $int$  be an interval in time. Then, the abstract formula

$ExecutionsStartedEndWithoutOverlapping(int, Ag, \mathcal{M})$

denotes that all actions executions started by the agent during  $int$  end in the interval, and there is no overlapping of executions during this interval.

Definition within the abstract language:

$ExecutionsStartedEndWithoutOverlapping(int, Ag, \mathcal{M}) \equiv$

$\forall A \in \text{ACTION} \forall t_1 \in int :$

$\text{ActionExStarts}(A, t_1, Ag, \mathcal{M}) \Rightarrow$

$\exists t_2 > t_1 : t_2 \in int \wedge$

$\text{ActionExEnds}(A, t_2, Ag, \mathcal{M}) \wedge$

$\text{NoOverlapDuringExecuting}(A, t_1, t_2, Ag, \mathcal{M})$

Let  $int$  be an interval in time. Then, the abstract formula

$ExecutionsEndedStartWithoutOverlapping(int, Ag, \mathcal{M})$

denotes that all actions executions ended in the agent during  $int$  have started in the interval, and there is no overlapping of executions during this interval.

Definition within the abstract language:

$ExecutionsEndedStartWithoutOverlapping(int, Ag, \mathcal{M}) \equiv$

$\forall A \in \text{ACTION} \forall t_2 \in int :$

$\text{ActionExEnds}(A, t_2, Ag, \mathcal{M}) \Rightarrow$

$\exists t_1 < t_2 : t_1 \in int \wedge$

$\text{ActionExStarts}(A, t_1, Ag, \mathcal{M}) \wedge$

$\text{NoOverlapDuringExecuting}(A, t_1, t_2, Ag, \mathcal{M})$

When, during some interval, every execution started during the interval ends without being overlapped and every execution ended has started without being overlapped, it is guaranteed that all executions in the interval don't overlap in any way. So:

Let  $int$  be an interval in time. Then, the abstract formula

$\text{NoOverlappingInWorld}(int, \mathcal{M})$

denotes that there is no overlapping of action executions in the world during  $int$ .

Definition within the abstract language:

$\text{NoOverlappingInWorld}(int, \mathcal{M}) \equiv$

$\text{ExecutionsStartedEndWithoutOverlapping}(int, EW, \mathcal{M}) \wedge$

$\text{ExecutionsEndedStartWithoutOverlapping}(int, EW, \mathcal{M})$

Let  $int$  be an interval in time. Then, the abstract formula

$\text{NoOverlappingInAgent}(int, \mathcal{M})$

denotes that there is no overlapping of action executions in the agent during  $int$ .

Definition within the abstract language:

$\text{NoOverlappingInAgent}(int, \mathcal{M}) \equiv$

$\text{ExecutionsStartedEndWithoutOverlapping}(int, Ag, \mathcal{M}) \wedge$

$\text{ExecutionsEndedStartWithoutOverlapping}(int, Ag, \mathcal{M})$

The following two abstract formulae are instantiations of the previous two formulae. By  $[0, \rightarrow)$ , the whole time line is denoted.

The abstract formula

$\text{NoOverlappingInWorld}(\mathcal{M})$  denotes that there is no overlapping of action executions in the world.

Definition within the abstract language:

$\text{NoOverlappingInWorld}(\mathcal{M}) \equiv \text{NoOverlappingInWorld}([0, \rightarrow], \mathcal{M})$

The abstract formula

$\text{NoOverlappingInAgent}(\mathcal{M})$  denotes that there is no overlapping of action executions in the agent.

Definition within the abstract language:

$\text{NoOverlappingInAgent}(\mathcal{M}) \equiv \text{NoOverlappingInAgent}([0, \rightarrow], \mathcal{M})$

### Concerning occasions and actions

As we explained above when we defined notions for occasions occurring and being observed, occasions are the world situations that can provide the agent with a reason to decide to do an action. If the agent in the generic architecture is a reactive agent, then it decides to do an action whenever it observes an occasion. For systems with a reactive agent it is thus vital that there aren't too many occasions, as this could result in overlapping action executions, which we want to exclude (COORD0). We define three notions here: the first one formalises that there is at most one occasion happening in the world between every two actions starting in a certain interval, and the second and third are agent and world variants formalising that there is exactly one occasion between every two actions starting in a certain interval. More precisely:

Let  $int$  be an interval in time. Then, the abstract formula

$\text{NotTooManyOccs}(int, \mathcal{M})$

denotes that

if there occurs an occasion before the first action starting during  $int$ ,

then it is the only occasion taking place before the first action and

if there occurs an occasion between two consecutive actions starting during  $int$ ,

then it is the only one between these two action beginnings.

Definition within the abstract language:

$\text{NotTooManyOccs}(int, \mathcal{M}) \equiv$

$[\forall A, B \in \text{ACTION} \forall t_1 \in int \forall t_0 \in [0, t_1) \cap int :$

$\text{FirstActionBegins}(A, t_1, int, EW, \mathcal{M}) \wedge$

$\oplus \text{Occ}(B, t_0, \mathcal{M}) \Rightarrow$

$\forall C \neq B \in \text{ACTION} : \neg \oplus \text{Occ}(C, t_0, \mathcal{M}) \wedge$

$\forall C \in \text{ACTION} \forall t \in ([0, t_1) \cap int) \setminus \{t_0\} :$

$\neg \oplus \text{Occ}(C, t, \mathcal{M})]$

$\wedge$

$[\forall A, B, C \in \text{ACTION} \forall t_2 \in int \forall t_3 \in [0, t_2) \cap int \forall t_4 \in [t_3, t_2) :$

$\text{ConsecutiveActionsBegin}(A, B, t_3, t_2, EW, \mathcal{M}) \wedge$

$\oplus \text{Occ}(C, t_4, \mathcal{M}) \Rightarrow$

$\forall D \neq C \in \text{ACTION} : \neg \oplus \text{Occ}(D, t_4, \mathcal{M}) \wedge$

$\forall D \in \text{ACTION} \forall t \in [t_3, t_2) \setminus \{t_4\} :$

$\neg \oplus \text{Occ}(D, t, \mathcal{M})]$

As before,  $[0, t_1) \cap int$  is the sub-interval of  $int$ , containing all time points before  $t_1$ . As in the definition above the first action in  $int$  starts at  $t_1$ , in this sub-interval of  $int$  at most one occasion is allowed to occur (at  $t_0$ ). Similarly, the interval  $[t_3, t_2)$  between two consecutive action beginnings in  $int$  contains at most one occasion (at  $t_4$ ), if  $\text{NotTooManyOccs}(int, \mathcal{M})$  holds.

Let  $int$  be an interval in time. Then, the abstract formula

$\text{OccsAndActionsAlternateInWorld}(int, \mathcal{M})$

denotes that

the first action starting in the world in  $int$  is preceded by its occasion occurring, and

this is the only occasion taking place before the first action

and

between every two consecutive action beginnings in  $int$ , an occasion for the later action occurs, and

this is the only occasion happening in the world between these two action beginnings.

$\text{OccsAndActionsAlternateInAgent}(int, \mathcal{M})$

denotes that

the first action the agent decides to do in  $int$  is preceded by its occasion being observed, and

this is the only occasion the agent observes before the first action

and

between every two consecutive decisions to do actions in  $int$ , the agent observed an occasion for the later action, and

and

this is the only occasion observed between these two action beginnings.

Definitions within the abstract language:

$\text{OccsAndActionsAlternateInWorld}(int, \mathcal{M}) \equiv$

$$\begin{aligned}
& [\forall A \in \text{ACTION} \forall t_1 \in int \\
& \text{FirstActionBegins}(A, t_1, int, EW, \mathcal{M}) \quad \Rightarrow \\
& \exists t_0 \in [0, t_1) \cap int : \oplus \text{Occ}(A, t_0, \mathcal{M}) \wedge \\
& \forall B \neq A \in \text{ACTION} : \neg \oplus \text{Occ}(B, t_0, \mathcal{M}) \wedge \\
& \forall B \in \text{ACTION} \forall t \in ([0, t_1) \cap int) \setminus \{t_0\} : \\
& \neg \oplus \text{Occ}(B, t, \mathcal{M})] \quad \wedge \\
& [\forall A, B \in \text{ACTION} \forall t_2 \in int \forall t_3 \in [0, t_2) \cap int : \\
& \text{ConsecutiveActionsBegin}(A, B, t_3, t_2, EW, \mathcal{M}) \Rightarrow \\
& \exists t_4 \in [t_3, t_2) : \oplus \text{Occ}(B, t_4, \mathcal{M}) \wedge \\
& \forall C \neq B \in \text{ACTION} : \neg \oplus \text{Occ}(C, t_4, \mathcal{M}) \wedge \\
& \forall C \in \text{ACTION} \forall t \in [t_3, t_2) \setminus \{t_4\} : \\
& \neg \oplus \text{Occ}(C, t, \mathcal{M})]
\end{aligned}$$

$$\begin{aligned}
\text{OccursAndActionsAlternateInAgent}(int, \mathcal{M}) \equiv & \\
& [\forall A \in \text{ACTION} \forall t_1 \in int : \\
& \text{FirstActionBegins}(A, t_1, int, \text{Ag}, \mathcal{M}) \Rightarrow \\
& \exists t_0 \in [0, t_1) \cap int : \oplus \text{ObsOcc}(A, t_0, \text{Ag}, \mathcal{M}) \wedge \\
& \forall B \neq A \in \text{ACTION} : \neg \oplus \text{ObsOcc}(B, t_0, \text{Ag}, \mathcal{M}) \wedge \\
& \forall B \in \text{ACTION} \forall t \in ([0, t_1) \cap int) \setminus \{t_0\} : \\
& \neg \oplus \text{ObsOcc}(B, t, \text{Ag}, \mathcal{M})] \wedge \\
& [\forall A, B \in \text{ACTION} \forall t_2 \in int \forall t_3 \in [0, t_2) \cap int : \\
& \text{ConsecutiveActionsBegin}(A, B, t_3, t_2, \text{Ag}, \mathcal{M}) \Rightarrow \\
& \exists t_4 \in [t_3, t_2) : \oplus \text{ObsOcc}(B, t_4, \text{Ag}, \mathcal{M}) \wedge \\
& \forall C \neq B \in \text{ACTION} : \neg \oplus \text{ObsOcc}(C, t_4, \text{Ag}, \mathcal{M}) \wedge \\
& \forall C \in \text{ACTION} \forall t \in [t_3, t_2) \setminus \{t_4\} : \\
& \neg \oplus \text{ObsOcc}(C, t, \text{Ag}, \mathcal{M})]
\end{aligned}$$

### Concerning the flow of information

In any agent system, pieces of information are transferred from one system component to another (from the agent to the world, or the other way around), and information is processed by the system components. This has to happen in an orderly manner. Pieces of information shouldn't overtake each other in links. Also, when the agent reasons, conclusions following from information that arrived in the agent later than other information should appear on the output later than the conclusions from the earlier information. And when the world produces observation results, these should occur in the same order as the facts that caused them. In order to phrase properties that formalise this, we need to *number* the moments that information arrives in (input or output) interfaces. The use of counting the moments that changes occur in the interfaces will become clear when we define the no-overtaking properties, in Subsection 3.6.5. We recursively define the notions we need. We start with the notions we need to phrase no-overtaking for links:

Let  $t_1$  be a moment in time and  $n \geq 1$  be a natural number. Then, the abstract formula

$\text{InputCount}(n, t_1, \text{Ag}, \mathcal{M})$

denotes that at  $t_1$ , information arrives in the input interface of the agent for the  $n$ th time, and

$\text{OutputCount}(n, t_1, \text{Ag}, \mathcal{M})$

denotes that at  $t_1$ , information arrives in the output interface of the agent for the  $n$ th time, and

$\text{InputCount}(n, t_1, \text{EW}, \mathcal{M})$

denotes that at  $t_1$ , information arrives in the input interface of the world for the  $n$ th time, and

$\text{OutputCount}(n, t_1, \text{EW}, \mathcal{M})$

denotes that at  $t_1$ , information arrives in the output interface of the world for the  $n$ th time.

Definition within the abstract language:

$n = 1 : \text{InputCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$

$\exists k_1 \in \text{groundliterals}(\text{AgInOnt}) :$   
 $\oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{Ag})) \models k_1 \wedge$   
 $\forall t < t_1 \forall k \in \text{groundliterals}(\text{AgInOnt}) :$   
 $\neg \oplus \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models k$

$n > 1 : \text{InputCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$

$\exists k_1 \in \text{groundliterals}(\text{AgInOnt}) :$   
 $\oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{Ag})) \models k_1 \wedge$   
 $\exists t_0 < t_1 \exists k_0 \in \text{groundliterals}(\text{AgInOnt}) :$   
 $\oplus \text{state}(\mathcal{M}, t_0, \text{input}(\text{Ag})) \models k_0 \wedge$   
 $\text{InputCount}(n - 1, t_0, \text{Ag}, \mathcal{M}) \wedge$   
 $\forall t \in \langle t_0, t_1 \rangle \forall k \in \text{groundliterals}(\text{AgInOnt}) :$   
 $\neg \oplus \text{state}(\mathcal{M}, t, \text{input}(\text{Ag})) \models k$

$n = 1 : \text{OutputCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$

$\exists k_1 \in \text{groundliterals}(\text{AgOutOnt}) :$   
 $\oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models k_1 \wedge$   
 $\forall t < t_1 \forall k \in \text{groundliterals}(\text{AgOutOnt}) :$   
 $\neg \oplus \text{state}(\mathcal{M}, t, \text{output}(\text{Ag})) \models k$

$n > 1 : \text{OutputCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$

$\exists k_1 \in \text{groundliterals}(\text{AgOutOnt}) :$   
 $\oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models k_1 \wedge$   
 $\exists t_0 < t_1 \exists k_0 \in \text{groundliterals}(\text{AgOutOnt}) :$   
 $\oplus \text{state}(\mathcal{M}, t_0, \text{output}(\text{Ag})) \models k_0 \wedge$   
 $\text{OutputCount}(n - 1, t_0, \text{Ag}, \mathcal{M}) \wedge$   
 $\forall t \in \langle t_0, t_1 \rangle \forall k \in \text{groundliterals}(\text{AgOutOnt}) :$   
 $\neg \oplus \text{state}(\mathcal{M}, t, \text{output}(\text{Ag})) \models k$

$$\begin{aligned}
n = 1 : \text{InputCount}(n, t_1, \text{EW}, \mathcal{M}) &\equiv \\
&\exists k_1 \in \text{groundliterals}(\text{AgOutOnt}) : \\
&\oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{EW})) \models k_1 \wedge \\
&\forall t < t_1 \forall k \in \text{groundliterals}(\text{AgOutOnt}) : \\
&\neg \oplus \text{state}(\mathcal{M}, t, \text{input}(\text{EW})) \models k \\
n > 1 : \text{InputCount}(n, t_1, \text{EW}, \mathcal{M}) &\equiv \\
&\exists k_1 \in \text{groundliterals}(\text{AgOutOnt}) : \\
&\oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{EW})) \models k_1 \wedge \\
&\exists t_0 < t_1 \exists k_0 \in \text{groundliterals}(\text{AgOutOnt}) : \\
&\oplus \text{state}(\mathcal{M}, t_0, \text{input}(\text{EW})) \models k_0 \wedge \\
&\text{InputCount}(n - 1, t_0, \text{EW}, \mathcal{M}) \wedge \\
&\forall t \in \langle t_0, t_1 \rangle \forall k \in \text{groundliterals}(\text{AgOutOnt}) : \\
&\neg \oplus \text{state}(\mathcal{M}, t, \text{input}(\text{EW})) \models k \\
n = 1 : \text{OutputCount}(n, t_1, \text{EW}, \mathcal{M}) &\equiv \\
&\exists k_1 \in \text{groundliterals}(\text{AgInOnt}) : \\
&\oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models k_1 \wedge \\
&\forall t < t_1 \forall k \in \text{groundliterals}(\text{AgInOnt}) : \\
&\neg \oplus \text{state}(\mathcal{M}, t, \text{output}(\text{EW})) \models k \\
n > 1 : \text{OutputCount}(n, t_1, \text{EW}, \mathcal{M}) &\equiv \\
&\exists k_1 \in \text{groundliterals}(\text{AgInOnt}) : \\
&\oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models k_1 \wedge \\
&\exists t_0 < t_1 \exists k_0 \in \text{groundliterals}(\text{AgInOnt}) : \\
&\oplus \text{state}(\mathcal{M}, t_0, \text{output}(\text{EW})) \models k_0 \wedge \\
&\text{OutputCount}(n - 1, t_0, \text{EW}, \mathcal{M}) \wedge \\
&\forall t \in \langle t_0, t_1 \rangle \forall k \in \text{groundliterals}(\text{AgInOnt}) : \\
&\neg \oplus \text{state}(\mathcal{M}, t, \text{output}(\text{EW})) \models k
\end{aligned}$$

We use the agent input and output ontologies  $\text{AgInOnt}$  and  $\text{AgOutOnt}$  for specifying possible atoms occurring in the interfaces of both the agent and the world. It might be surprising that we don't use world ontologies in the definition of  $\text{InputCount}(n, t_1, \text{EW}, \mathcal{M})$  and  $\text{OutputCount}(n, t_1, \text{EW}, \mathcal{M})$ . The reason for this is that we didn't define specific interface ontologies for the world. The input ontology of the world is exactly the same as the output ontology of the agent. The output ontology of the world slightly differs from the input ontology of the agent, as the world also outputs new world facts, based on the ontology world info. But as information of this last type isn't transferred to the agent through the information link, the only interesting literals are those of  $\text{AgInOnt}$ .

To specify the no-overtaking property for the reasoning of the agent, we have to define a notion to count the moments that the agent observes occasions, as only this agent input information can lead to decisions to do actions. We

also define a notion for counting the moments that the agent decides to do actions. Strictly speaking, this is not necessary, as the only output atoms that the agent generates are decisions to do actions, so we could just as well use  $\text{OutputCount}(n, t, \text{Ag}, \mathcal{M})$  for this purpose. The only reason for defining a new notion is aesthetics.

Let  $t_1$  be a moment in time and  $n \geq 1$  be a natural number. Then, the abstract formula

$\text{ObsOccCount}(n, t_1, \text{Ag}, \mathcal{M})$	denotes that at $t_1$ , the agent receives information about observed occasions for the $n$ th time, and
$\text{ActionCount}(n, t_1, \text{Ag}, \mathcal{M})$	denotes that at $t_1$ , the agent decides to do an action (or actions) for the $n$ th time.

Definition within the abstract language:

$n = 1 : \text{ObsOccCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$	$\exists A \in \text{ACTION} :$ $\oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M}) \wedge$ $\forall t < t_1 \forall B \in \text{ACTION} :$ $\neg \oplus \text{ObsOcc}(B, t, \text{Ag}, \mathcal{M})$
$n > 1 : \text{ObsOccCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$	$\exists A \in \text{ACTION} :$ $\oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M}) \wedge$ $\exists t_0 < t_1 \exists B \in \text{ACTION} :$ $\oplus \text{ObsOcc}(B, t_0, \text{Ag}, \mathcal{M}) \wedge$ $\text{ObsOccCount}(n-1, t_0, \text{Ag}, \mathcal{M}) \wedge$ $\forall t \in \langle t_0, t_1 \rangle \forall C \in \text{ACTION} :$ $\neg \oplus \text{ObsOcc}(C, t_0, \text{Ag}, \mathcal{M})$
$n = 1 : \text{ActionCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$	$\exists A \in \text{ACTION} :$ $\text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) \wedge$ $\forall t < t_1 \forall B \in \text{ACTION} :$ $\neg \text{ActionExStarts}(B, t, \text{Ag}, \mathcal{M})$
$n > 1 : \text{ActionCount}(n, t_1, \text{Ag}, \mathcal{M}) \equiv$	$\exists A \in \text{ACTION} :$ $\text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) \wedge$ $\exists t_0 < t_1 \exists B \in \text{ACTION} :$ $\text{ActionExStarts}(B, t_0, \text{Ag}, \mathcal{M}) \wedge$ $\text{ActionCount}(n-1, t_0, \text{Ag}, \mathcal{M}) \wedge$ $\forall t \in \langle t_0, t_1 \rangle \forall C \in \text{ACTION} :$ $\neg \text{ActionExStarts}(C, t, \text{Ag}, \mathcal{M})$

Finally, we need notions to specify the no-overtaking property for the generation of observation results from facts happening in the world. Not all facts happening are relevant; we only will be interested in occasions for actions oc-

curing, and the matching observation results. We define a notion for counting the moments that new occasions happen in the world, and a notion for counting the moments that the world generates observations of these occasions.

Let  $t_1$  be a moment in time and  $n \geq 1$  be a natural number. Then, the abstract formula

$\text{OccCount}(n, t_1, \text{EW}, \mathcal{M})$

denotes that at  $t_1$ , one or more occasions occur in the world for the  $n$ th time, and

$\text{ObsOccCount}(n, t_1, \text{EW}, \mathcal{M})$

denotes that at  $t_1$ , the world generates information about observed occasions for the  $n$ th time.

Definition within the abstract language:

$n = 1 : \text{OccCount}(n, t_1, \text{EW}, \mathcal{M}) \equiv$

$\exists A \in \text{ACTION} :$   
 $\oplus \text{Occ}(A, t_1, \mathcal{M}) \wedge$   
 $\forall t < t_1 \forall B \in \text{ACTION} :$   
 $\neg \oplus \text{Occ}(B, t, \mathcal{M})$

$n > 1 : \text{OccCount}(n, t_1, \text{EW}, \mathcal{M}) \equiv$

$\exists A \in \text{ACTION} :$   
 $\oplus \text{Occ}(A, t_1, \mathcal{M}) \wedge$   
 $\exists t_0 < t_1 \exists B \in \text{ACTION} :$   
 $\oplus \text{Occ}(B, t_0, \mathcal{M}) \wedge$   
 $\text{OccCount}(n - 1, t_0, \text{EW}, \mathcal{M}) \wedge$   
 $\forall t \in \langle t_0, t_1 \rangle \forall C \in \text{ACTION} :$   
 $\neg \oplus \text{Occ}(C, t_0, \mathcal{M})$

$n = 1 : \text{ObsOccCount}(n, t_1, \text{EW}, \mathcal{M}) \equiv$

$\exists A \in \text{ACTION} :$   
 $\oplus \text{ObsOcc}(A, t_1, \text{EW}, \mathcal{M}) \wedge$   
 $\forall t < t_1 \forall B \in \text{ACTION} :$   
 $\neg \oplus \text{ObsOcc}(B, t, \text{EW}, \mathcal{M})$

$n > 1 : \text{ObsOccCount}(n, t_1, \text{EW}, \mathcal{M}) \equiv$

$\exists A \in \text{ACTION} :$   
 $\oplus \text{ObsOcc}(A, t_1, \text{EW}, \mathcal{M}) \wedge$   
 $\exists t_0 < t_1 \exists B \in \text{ACTION} :$   
 $\oplus \text{ObsOcc}(B, t_0, \text{EW}, \mathcal{M}) \wedge$   
 $\text{ObsOccCount}(n - 1, t_0, \text{EW}, \mathcal{M}) \wedge$   
 $\forall t \in \langle t_0, t_1 \rangle \forall C \in \text{ACTION} :$   
 $\neg \oplus \text{ObsOcc}(C, t_0, \text{EW}, \mathcal{M})$

## 3.6 Basic properties

In the previous section, we have defined predicates of AL, which we will use (and have used in Section 3.4) to phrase properties. We need a number of properties which describe requirements on the basic behaviour of the agent and the world and the information transferral between them. In the proofs made for this chapter, these properties are part of the premises of the proofs. By their general nature, the basic properties will be needed in the proofs of many other properties besides action successfulness. The properties can often be proved directly from the semantics of the design specification. We won't go into these proofs here. Five classes of properties exist.

### 3.6.1 Properties concerning actions starting and ending

There are two basic demands made on the world regarding the action executions starting and ending. These are that every start of an execution of action A has to be followed by an end of such an execution, and that every end of an execution of A may only appear when there has been an execution of A starting earlier.

$$\begin{aligned} \text{END1 :} \\ \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_1 : \\ \quad \text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M}) \quad \Rightarrow \\ \exists t_2 > t_1 : \text{ActionExEnds}(A, t_2, \text{EW}, \mathcal{M}) \end{aligned}$$

$$\begin{aligned} \text{END2 :} \\ \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_2 : \\ \quad \text{ActionExEnds}(A, t_2, \text{EW}, \mathcal{M}) \quad \Rightarrow \\ \exists t_1 < t_2 : \text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M}) \end{aligned}$$

Note that these properties don't state that for every start of an action there is a matching end. For example, these properties allow ten consecutive starts of executions of A, followed by a single end. COORD0 does imply that there is one matching end for every action starting.

### 3.6.2 Rationality properties

When the agent in the system satisfies the rationality properties presented in this section, this means that it makes the right decisions to perform actions. More specifically, there are three properties.

RAT1 states that the agent will always decide to do an action when it observes an occasion for the action.

$$\begin{aligned}
&\text{RAT1 :} \\
&\forall \mathcal{M} \in \text{Traces}(\text{Ag}) \forall A \in \text{ACTION} \forall t_1 : \\
&\quad \oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\
&\exists t_2 > t_1 : \text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M})
\end{aligned}$$

We call the agent that always acts when it sees an occasion *reactive*, as we stated before. RAT1 above formalises the reasoning behaviour of reactive agents. But the agent could be more careful, and decide to do an action when it observes an occasion for the action *and* it is not still busy with another action execution. So, this more deliberative agent doesn't simply decide to do an action whenever it observes an occasion for it; only when earlier actions have been fully executed, a new action can be started. For this class, we could formalise a variant of RAT1.

We use different properties for formalising the reasoning of *subclasses* of agents, because for different subclasses, there are different ways to prove COORD0 (which states that action executions don't overlap) and COORD5 (which states that actions are applicable at the time they start in the world). If we know that the agent always acts when it has occasion for this, we need properties phrasing that the world doesn't provide occasions too frequently. When the agent has a different acting policy, different properties need to be phrased. If the agent in the generic architecture is of the more deliberative type, then less strict demands are needed on the dynamics of the world in order to guarantee COORD0 and COORD5. Summarising, we could say that if the agent is more smart, the world can be more wild. If we wouldn't create subclasses of agents, then we would have to use the most strict demands on the dynamics of the world in order to prove COORD0 and COORD5.

RAT2 states that when an agent decides to do an action it must have observed an occasion. This excludes behaviour in which the agent randomly decides to do things, which would certainly invalidate COORD0. RAT2 should hold for all agents that are instantiations of the generic architecture; no separate property variants for subclasses of agents are needed here. This is the formalisation of RAT2:

$$\begin{aligned}
&\text{RAT2 :} \\
&\forall \mathcal{M} \in \text{Traces}(\text{Ag}) \forall A \in \text{ACTION} \forall t_2 : \\
&\quad \text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\
&\exists t_1 < t_2 : \oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M})
\end{aligned}$$

RAT3 says that occasionality implies applicability. This last property is necessary, because the agent could have wrong knowledge on when actions are executable. For example, if the agent is a pedestrian walking in busy city traffic, it may have the knowledge: ‘When the traffic light goes red, I decide to perform the action of crossing the street’. So, in this system, an occasion for the action of crossing the street is a red traffic light. But this is not at all an applicable situation: in the situation that the traffic light is red, the action of crossing the street will probably not be executed successfully. Assuming the agent knows about the signals of traffic lights and their meaning, the agent is not rational in its knowledge concerning occasions.

In contrast to the other two rationality properties, RAT3 is a property of the world, as applicability and occasionality are notions defined in the context of the world.

$$\begin{aligned}
 \text{RAT3 :} \\
 \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t : \\
 \quad \text{Occ}(A, t, \mathcal{M}) \quad \Rightarrow \\
 \quad \text{App1}(A, t, \mathcal{M})
 \end{aligned}$$

### 3.6.3 Properties concerning facts and observations

The factual world information is assumed to be two-valued; this is laid down in 2VAL, which we phrase in the detailed language TL instead of AL:

$$\begin{aligned}
 \text{2VAL :} \\
 \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall a \in \text{groundatoms}(\text{world info}) \forall t : \\
 \quad \text{state}(\mathcal{M}, t, \text{output}(\text{EW})) \models a \quad \vee \\
 \quad \text{state}(\mathcal{M}, t, \text{output}(\text{EW})) \models \neg a
 \end{aligned}$$

All factual world information will be observed, and all observed information corresponds to world facts. To state this, there are properties OBS1 and OBS2:

$$\begin{aligned}
 \text{OBS1 :} \\
 \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall \varphi \in \text{SL}(\text{world info}) \forall t_1 : \\
 \quad \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models \varphi \quad \Rightarrow \\
 \exists t_2 \geq t_1 : \oplus \text{state}(\mathcal{M}, t_2, \text{output}(\text{EW})) \models \text{observation\_result}(\varphi) \\
 \text{OBS2:} \\
 \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall \varphi \in \text{SL}(\text{world info}) \forall t_2 : \\
 \quad \oplus \text{state}(\mathcal{M}, t_2, \text{output}(\text{EW})) \models \text{observation\_result}(\varphi) \quad \Rightarrow \\
 \exists t_1 \leq t_2 : \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models \varphi
 \end{aligned}$$

Things don't happen without a reason in the world. In our generic architecture, a change is always either the effect of an action or of an event. Property CHANGE formalises this.

$$\begin{array}{l}
\text{CHANGE :} \\
\forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall l \in \text{groundliterals}(\text{world info}) \forall t : \\
\quad \text{Happens}(l, t, \mathcal{M}) \\
(\exists A \in \text{ACTION} : \text{ActionEff}(A, l, t, \mathcal{M}) \quad \vee \\
\quad \text{EventEff}(l, t, \mathcal{M})) \quad \Rightarrow
\end{array}$$

This property isn't a demand on the world, as is usual with world properties. Instead, CHANGE follows easily from the definitions of ActionEff and EventEff.

### 3.6.4 Properties of interaction

As can be seen in Figure 3.1, there is information transfer from the world to the agent and from the agent to the world, respectively. This transferral of information needs to function properly. This comes down to three demands:

- \* Information at the source must be present at the destination some time later.
- \* Information at the destination must have been present at the source some time earlier.
- \* Pieces of information do not overtake each other while being transferred.

We will formalise the first two demands for the two links in the generic architecture that facilitate information transfer. This results in four properties. The properties for no-overtaking in links are defined in the next subsection. We start with the properties for information transfer from the world to the agent, followed by the properties for information transfer from the agent to the world.

$$\begin{array}{l}
\text{INT1 :} \\
\forall \mathcal{M} \in \text{Traces}(\text{S}) \forall l \in \text{AgInOnt} \forall t_1 : \\
\quad \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models l \quad \Rightarrow \\
\exists t_2 > t_1 : \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models l
\end{array}$$

$$\begin{aligned}
&\text{INT2 :} \\
&\forall \mathcal{M} \in \text{Traces}(\mathbf{S}) \forall I \in \text{AgInOnt} \forall t_2 : \\
&\quad \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models I \quad \Rightarrow \\
&\exists t_1 < t_2 : \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models I \\
&\text{INT3 :} \\
&\forall \mathcal{M} \in \text{Traces}(\mathbf{S}) \forall I \in \text{AgOutOnt} \forall t_1 : \\
&\quad \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models I \quad \Rightarrow \\
&\exists t_2 > t_1 : \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{EW})) \models I \\
&\text{INT4 :} \\
&\forall \mathcal{M} \in \text{Traces}(\mathbf{S}) \forall I \in \text{AgOutOnt} \forall t_2 : \\
&\quad \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{EW})) \models I \quad \Rightarrow \\
&\exists t_1 < t_2 : \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models I
\end{aligned}$$

These properties state that every literal put on a link at the source will arrive at the destination some time later, at that every literal arriving at the destination of the link was present at the source some time earlier.

Like the properties 2VAL, OBS1 and OBS2 in Section 3.6.3, the properties above are phrased in TL, instead of in AL. The reason for this is that these properties describe basic, very low-level behaviour features. Thus, the nature of these properties fits better with the abstraction level of TL than with the level of AL. Also, there seems to be no obvious manner to define AL notions in order to simplify these properties.

In the phrasing of the interaction properties, we assume that only one information link arrives in the input interface of each system component. This holds for the generic architecture at which we focus in this chapter, but other system architectures exist which are different in this respect. For example, in systems with multiple agents which all communicate with each other, a link arrives in an agent's input interface for each of its fellow agents. In systems like these, information arriving in an input interface can come from different sources, and thus part of the interaction properties will be structurally different from the interaction properties given above. If we look at the properties given here, and imagine that two information links arrive in the agent, this leads to a modified property INT2, which states that information arriving in the agent must have come either from the one source or from the other source. Property INT1 would not be affected by the multiple links; we would use one property like this for each link.

In the literature on file transmission, interaction properties like ours have been formally studied. In [54, 115] knowledge-based transmission protocols are presented that are proven to be correct with respect to these properties.

### 3.6.5 No-overtaking properties

It is essential that pieces of information don't overtake each other, neither in links nor while being processed by a system component. We give four properties that phrase this no-overtaking restriction, for the two links of the generic architecture, for the information processing of the agent and for the generation of observation results of occasions happening in the world. For this, we use the notions for numbering the moments in time that information arrives in interfaces of the agent and the world which we defined at the end of Section 3.5.

In the link properties below, we formally express that information that arrives in an output interface at the moment numbered  $n$  should arrive in the appropriate input interface at the corresponding moment numbered  $n$ , and vice versa. Some time elapses between these moments, as information transferral takes time. The first property formalises no-overtaking for the link from the world to the agent, and the second property does the same for the link from the agent to the world. In all properties in this subsection,  $n$  is a positive natural number.

NOOV1 :

$$\begin{aligned}
 & \forall \mathcal{M} \in \text{Traces}(\mathbf{S}) \ \forall \mathbf{k} \in \text{groundliterals}(\text{AgInOnt}) \ \forall t_1 \ \forall t_2 > t_1 : \\
 & \quad [\oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models \mathbf{k} \ \wedge \\
 & \quad \text{OutputCount}(n, t_1, \text{EW}, \mathcal{M}) \ \wedge \\
 & \quad \text{InputCount}(n, t_2, \text{Ag}, \mathcal{M}) \ \Rightarrow \\
 & \quad \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models \mathbf{k}] \ \wedge \\
 & \quad [\oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{Ag})) \models \mathbf{k} \ \wedge \\
 & \quad \text{InputCount}(n, t_2, \text{Ag}, \mathcal{M}) \ \wedge \\
 & \quad \text{OutputCount}(n, t_1, \text{EW}, \mathcal{M}) \ \Rightarrow \\
 & \quad \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})) \models \mathbf{k}]
 \end{aligned}$$

NOOV2 :

$$\begin{aligned}
 & \forall \mathcal{M} \in \text{Traces}(\mathbf{S}) \ \forall \mathbf{k} \in \text{groundliterals}(\text{AgOutOnt}) \ \forall t_1 \ \forall t_2 > t_1 : \\
 & \quad [\oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models \mathbf{k} \ \wedge \\
 & \quad \text{OutputCount}(n, t_1, \text{Ag}, \mathcal{M}) \ \wedge \\
 & \quad \text{InputCount}(n, t_2, \text{EW}, \mathcal{M}) \ \Rightarrow \\
 & \quad \oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{EW})) \models \mathbf{k}] \ \wedge \\
 & \quad [\oplus \text{state}(\mathcal{M}, t_2, \text{input}(\text{EW})) \models \mathbf{k} \ \wedge \\
 & \quad \text{InputCount}(n, t_2, \text{EW}, \mathcal{M}) \ \wedge \\
 & \quad \text{OutputCount}(n, t_1, \text{Ag}, \mathcal{M}) \ \Rightarrow \\
 & \quad \oplus \text{state}(\mathcal{M}, t_1, \text{output}(\text{Ag})) \models \mathbf{k}]
 \end{aligned}$$

Now, it is impossible that pieces of information overtake each other in a link. This can easily be explained. Suppose a literal  $k_1$  arrives in an output interface at  $t_1$ , and that the later literal  $k_2$  arrives there at  $t_2$ , with  $t_2 > t_1$ . If we have  $\text{OutputCount}(n_1, t_1, \dots)$  and  $\text{OutputCount}(n_2, t_2, \dots)$ , then we know that  $n_2 > n_1$ . Suppose  $t_3$  and  $t_4$  are the corresponding input moments for the interface at the other side of the link, that is,  $\text{InputCount}(n_1, t_3, \dots)$  and  $\text{InputCount}(n_2, t_4, \dots)$ . Then the literals  $k_1$  and  $k_2$  arrive at  $t_3$  and  $t_4$ , respectively, according to the properties above. Because time  $t_4$  has a higher number than  $t_3$ ,  $t_4$  is later than  $t_3$  and  $k_2$  arrives later than  $k_1$ , as it should.

As with the properties for interaction in the previous subsection, we need variants of these properties in case more links arrive in the input interface of a system component. We don't elaborate on this in this chapter.

We now give the no-overtaking property for the information processing of the agent. For this property, we use the notions  $\text{ObsOccCount}(\dots, \text{Ag}, \dots)$  and  $\text{ActionCount}(\dots, \text{Ag}, \dots)$  instead of  $\text{InputCount}(\dots, \text{Ag}, \dots)$  and  $\text{OutputCount}(\dots, \text{Ag}, \dots)$ , because not all information arriving in the input of the agent has to lead to new conclusions, while links do transfer each new bit of information that appears in the output interface at the start of the link. So, it is possible that an observed occasion at  $\text{InputCount}(4, t_1, \text{Ag}, \mathcal{M})$  leads to the decision to do an action at  $\text{OutputCount}(2, t_2, \text{Ag}, \mathcal{M})$ , where  $t_2$  must still be later than  $t_1$ , of course. We have the following property:

$$\begin{aligned}
& \text{NOOV3 :} \\
& \forall \mathcal{M} \in \text{Traces}(\text{Ag}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
& \quad [\oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ObsOccCount}(n, t_1, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ActionCount}(n, t_2, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\
& \quad \text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M})] \quad \wedge \\
& \quad [\text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ActionCount}(n, t_2, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ObsOccCount}(n, t_1, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\
& \quad \oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M})]
\end{aligned}$$

Agents that satisfy this property must reason quick enough to prevent new information from being overwritten by even newer information before it is processed. Suppose the agent doesn't pay attention to incoming information during the time it is reasoning. If there are two or more moments that new information arrives in the agent's input during the period that it is reasoning, then the agent will process the new information in the next round of reasoning, and the conclusions from both pieces of information will appear in the output of the agent at the same time. Situations like this are excluded by NOOV3.

Finally, we give the no-overtaking property for the generation of observed occasions by the world. For this property, we make use of the notions  $\text{OccCount}(\dots, \text{EW}, \dots)$  and  $\text{ObsOccCount}(\dots, \text{EW}, \dots)$ :

$$\begin{aligned}
& \text{NOOV4 :} \\
& \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
& \quad [\oplus \text{Occ}(A, t_1, \mathcal{M}) \wedge \\
& \quad \text{OccCount}(n, t_1, \text{EW}, \mathcal{M}) \wedge \\
& \quad \text{ObsOccCount}(n, t_2, \text{EW}, \mathcal{M})] \quad \Rightarrow \\
& \quad [\oplus \text{ObsOcc}(A, t_2, \text{EW}, \mathcal{M})] \wedge \\
& \quad [\oplus \text{ObsOcc}(A, t_2, \text{EW}, \mathcal{M}) \wedge \\
& \quad \text{ObsOccCount}(n, t_2, \text{EW}, \mathcal{M})] \wedge \\
& \quad \text{OccCount}(n, t_1, \text{EW}, \mathcal{M}) \quad \Rightarrow \\
& \quad \oplus \text{Occ}(A, t_1, \mathcal{M})]
\end{aligned}$$

### 3.7 Proving COORD0

In this section, we will give a proof of COORD0:

$$\begin{aligned}
& \text{COORD0 :} \\
& \forall \mathcal{M} \in \text{Traces}(\text{S}) : \\
& \quad \text{NoOverlappingInWorld}(\mathcal{M}) \wedge \\
& \quad \text{NoOverlappingInAgent}(\mathcal{M})
\end{aligned}$$

As we hinted on earlier, the proof of COORD0 we present in this section is suitable for a subclass of the generic architecture from Section 3.2. We prove COORD0 for the subclass of systems with a *reactive* agent. We would have liked this proof to be done once and for all, for all possible systems in the class. By identifying a set of generic demands on the agent and a set of demands on the world and giving a proof of COORD0 based on these demands, all that would remain to be done when verifying COORD0 for some specific system is proving that all demands on the specific world and agent are fulfilled. The reason that we don't give a totally generic proof for COORD0 is that there are different kinds of agents (reactive, deliberative, ...). There is a conflict between finding generic properties that are strong enough to prove COORD0 and weak enough to allow as much dynamic interaction between agent and world as is possible, considered the specific design of the agent. Therefore, we think it is better to prove COORD0 for classes of systems with agents that are similar in the way they make decisions.

All reactive systems (which are systems having a reactive agent) are characterised by a large mutual dependency between the agent and the world. The world immediately reacts to information coming from the agent (that is, actions to be performed) and the agent also immediately reacts to information coming from the world (observation results, caused by actions or events). There are no thresholds of deliberation in the world or the agent; as soon as the agent sees an occasion for an action, it decides to perform the action, without further consideration about past actions or its resources or other matters that could cause postponement of the action. This is why we call such systems reactive: there is as little reasoning as possible, observations directly imply actions to be performed. The world basically works the same way: when an action to be performed arrives in the world, the world simply starts execution of the action, without checking whether there is still some other execution going on. In the light of COORDO, systems like this are very fragile. As soon as there is one event at an undesirable moment in time, COORDO doesn't hold anymore. The behaviours of the world and the agent fit into each other like a zipper: each relevant step on one side (an occasion arising, or a decision to do an action) must be followed by exactly one appropriate step on the other side (a decision to do an action, or an occasion arising for the next action).

As long as both parties only react to each other in some constrained way, the no-overlapping of action executions can be guaranteed. The central feature that all correctly working reactive systems have in common is that occasions for actions are observable to the agent at exactly the right moments: they are not allowed to happen too often or too little. Before the agent decides to do its first action, there must have been exactly one observed occasion, namely the one that led the agent to its decision to do the first action. And for every two consecutive decisions to do actions, the agent must have observed exactly one occasion at some moment in between, namely the occasion for the second action of the pair. When these demands are fulfilled, there never is some unprocessed observed occasion left over that can lead to a to-be-performed-atom at an undesirable moment.

But it is not only important that observations of occasions alternate exactly with decisions to perform actions; it is also crucial that these occasions are observed at the right time. When the agent observes an occasion for a new action while it is still in the middle of an action execution, this may lead the agent to deciding to do the new action before the current action execution has ended. Non-overlapping of action executions can be guaranteed if the agent only observes occasions while it is not executing any action. The agent only observes an occasion when some time earlier an occasion has arisen in the world. This means that the state of the world has changed, which is caused

by an action or an event. So, effects of both events and actions have to be constrained in their occurrence. In the properties we present in Subsection 3.7.1, we will forbid events during action executions, and demand that new occasions caused by actions always happen at the end of action executions. To explain the latter demand, if during an action execution one of the earlier action effects of the action already leads to an occasion for some new action, the agent may decide to do another action during the execution of the current one. By demanding that during an action execution the agent only observes occasions at the end of the execution, we prevent this.

We use induction to establish a proof of `COORD0`. The reason for using induction is that induction allows you to focus on a slice of the time-line the property applies to and to make strong assumptions regarding the validity of the property during the rest of the time. In this case, we use induction over the number of action executions. Focusing on the  $n$ th action execution, all executions taking place earlier are assumed to be well behaved (that is, `COORD0` holds for them). This is (part of) the induction hypothesis. In the induction step, only the results for the  $n$ th execution have to be proved, assuming all earlier executions are proper. Actually, we have an induction hypothesis which is stronger than just non-overlapping for earlier executions; the hypothesis also incorporates that occasions and actions alternate, as this is necessary to prove non-overlapping.

In order to make the induction step, a number of properties of the agent and of the world, respectively, are needed. They are used in the proof to make the induction step, so they have conditions that state that earlier executions were non-overlapping or that earlier occasions and actions alternate.

### 3.7.1 Properties needed for proving `COORD0`

Here, we present the properties needed to prove `COORD0`. They constitute assumptions on the behaviour of the agent and the world.

First, there is a property describing the demand on agent behaviour in order to guarantee non-overlapping. Informally, this property states:

‘When the agent decides to do action A at  $t_1$   
and  
the agent decides to do action B at a later time  $t_2$   
and  
occasions and actions alternate in the agent up to  $t_1$   
then

the agent must have observed an occasion for B between  $t_1$  and  $t_2$ .’

This means that the agent never decides to do an action without an observed occasion, and that these occasions must be observed in between the decisions to do actions. Formally:

$$\begin{aligned}
 \text{ORDER1 :} \\
 \forall \mathcal{M} \in \text{Traces}(\text{Ag}) \ \forall A, B \in \text{ACTION} \ \forall t_1 \ \forall t_2 \geq t_1 : \\
 \quad \text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) & \quad \wedge \\
 \quad \text{ActionExStarts}(B, t_2, \text{Ag}, \mathcal{M}) & \quad \wedge \\
 \quad \text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) & \quad \Rightarrow \\
 \exists t_3 \in [t_1, t_2) : \oplus \text{ObsOcc}(B, t_3, \text{Ag}, \mathcal{M})
 \end{aligned}$$

Next, we have a number of properties that phrase demands on the behaviour of the world. Some properties describe the world behaving naturally; for example, one property forbids the world to produce ‘wild’ ended-atoms, that are not preceded by a matching to\_be\_performed-atom. Other properties constrain the dynamics of the world. We start with the property we just mentioned:

‘When action A ends at  $t_1$  in the world  
and  
action B ends in the world later, at  $t_2$   
and  
action executions don’t overlap in the world up to  $t_1$   
then  
action B must have started in the world between  $t_1$  and  $t_2$ ’

And this is the formal form:

$$\begin{aligned}
 \text{ORDER2 :} \\
 \forall \mathcal{M} \in \text{Traces}(\text{EW}) \ \forall A, B \in \text{ACTION} \ \forall t_1 \ \forall t_2 > t_1 : \\
 \quad \text{ActionExEnds}(A, t_1, \text{EW}, \mathcal{M}) & \quad \wedge \\
 \quad \text{ActionExEnds}(B, t_2, \text{EW}, \mathcal{M}) & \quad \wedge \\
 \quad \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) & \quad \Rightarrow \\
 \exists t_3 \in \langle t_1, t_2 \rangle : \text{ActionExStarts}(B, t_3, \text{EW}, \mathcal{M})
 \end{aligned}$$

The second world property demands that in the world at most one occasion occurs preceding each action starting. This means that when there is an action

execution at some moment, which doesn't yield effects that cause an occasion for a new action to occur, at most one event is allowed to take place that does lead to an occasion before the next action. And in case the earlier action execution gives rise to a new occasion, no events leading to other occasions should take place. This is the informal form of ORDER3:

'When action executions don't overlap in the world until  $t_1$   
then  
there are not too many occasions occurring in the world up to  $t_1$ .'

The formalisation of this property is:

$$\begin{aligned} \text{ORDER3 :} \\ \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall t_1 : \\ \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) \Rightarrow \\ \text{NotTooManyOccs}([0, t_1], \mathcal{M}) \end{aligned}$$

The next world property forbids events happening during action executions. As we explained in the informal account of the properties needed at the start of this section, it is not only important not to have too many occasions happening (ORDER3), but also that these occasions don't happen during action executions, as the agent could then decide to do another action before the current execution is over. Properties ORDER4 and ORDER5 relate to this. As occasions arise as a consequence of either an action or of an event, we have to make demands regarding both events and actions, as we do in ORDER4 and ORDER5, respectively.

Informally, this is ORDER4:

'When there is an action execution in the world from  $t_1$  to  $t_2$   
and  
action executions don't overlap until  $t_1$   
then  
no events happen during the action execution.'

Formally:

$$\begin{aligned} \text{ORDER4 :} \\ \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\ \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \wedge \\ \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) \quad \Rightarrow \\ \text{NoEventsDuring}([t_1, t_2], \mathcal{M}) \end{aligned}$$

Note that we have another property forbidding events during action executions, namely COORD3. COORD3 is not usable for proving COORD0, as one of the conditions of COORD3 is that there is no overlapping of action executions in the world, and this is (half of) COORD0.

The last property we need for proving COORD0 states that occasions that happen during action executions and which are caused by actions always become observed at the same time an action execution ends. This implies that when during an action execution the agent observes an occasion, this is either at the end of this execution, or at the end of another execution which is overlapping with the current execution. During the proof of COORD0, the second case will be excluded. This is the informal form of ORDER5:

‘When there is an action execution in the world from  $t_1$  to  $t_2$   
and  
action executions don’t overlap in the world until  $t_1$   
and  
occasions and actions alternate up to  $t_1$   
and  
no events happen during the action execution  
and  
an occasion becomes observable in the world at a moment  $t_3$   
during the execution  
then  
there is an action execution ending at  $t_3$ .’

And this is the formal form of this property:

ORDER5 :

$$\begin{aligned}
\forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A, B \in \text{ACTION} \forall t_1 \forall t_2 > t_1 \forall t_3 \in [t_1, t_2] : \\
& \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) && \wedge \\
& \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) && \wedge \\
& \text{OccsAndActionsAlternateInWorld}([0, t_1], \mathcal{M}) && \wedge \\
& \text{NoEventsDuring}([t_1, t_2], \mathcal{M}) && \wedge \\
& \oplus \text{ObsOcc}(B, t_3, \text{EW}, \mathcal{M}) && \Rightarrow \\
\exists C \in \text{ACTION} : \text{ActionExEnds}(C, t_3, \text{EW}, \mathcal{M})
\end{aligned}$$

This property is rather complex, so we will provide some more explanation. The property focuses on the execution of A from  $t_1$  to  $t_2$ , and assumes that the

induction hypothesis (that action executions don't overlap and occasions and actions alternate) holds for the period before the action execution. Also, the property has a condition that no events take place during the execution. Now, if an occasion for action B becomes observable in the world *during* the execution, the occasion that caused the observation must also have happened during the execution of A. This is so because

$\text{OccsAndActionsAlternateInWorld}([0, t_1], \mathcal{M})$  makes it impossible that this occasion takes place *before*  $t_1$ , as the occasions for all preceding actions as well as for A itself already happened before  $t_1$ , alternating with the beginnings of the actions. As the occasion takes place during the execution of A, it must be the effect of an action, because no events take place during the execution. Now, the demand that ORDER5 makes is that the effects of this action (the occasion for B) always become observable at the end of the action execution.

### 3.7.2 The proof of COORD0

As explained before, we do the proof of COORD0 by induction on the number of action executions. Actually, we prove a theorem which is stronger than COORD0: we load the induction. We use an induction hypothesis which states that action executions don't overlap until the beginning of the  $n$ th execution, and that occasions and actions alternate until this same moment, and in the induction step we prove that this hypothesis also holds until the start of the  $n+1$ th execution.

The proof is illustrated in various figures throughout the rest of this subsection. We start the proof by repeating COORD0:

$$\begin{aligned} \forall \mathcal{M} \in \text{Traces}(S) : \\ \text{NoOverlappingInWorld}(\mathcal{M}) \quad \wedge \\ \text{NoOverlappingInAgent}(\mathcal{M}) \end{aligned}$$

In order to prove this, we use the following induction hypothesis:

$$\begin{aligned} \text{IH} : \\ \text{NoOverlappingInAgent}([0, t_1], \mathcal{M}) \quad \wedge \\ \text{NoOverlappingInWorld}([0, t_2], \mathcal{M}) \quad \wedge \\ \text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) \quad \wedge \\ \text{OccsAndActionsAlternateInWorld}([0, t_2], \mathcal{M}) \end{aligned}$$

Here,  $t_1$  and  $t_2$  are the moments in time at which the  $n$ th action execution begins in the agent and in the world, respectively. We now start the proof.

Take a trace  $\mathcal{M}$  of  $S$ . The **induction basis** is the case  $n = 0$ ; we want to prove that until the start of the first action all is well.

Suppose the first action execution starts at time  $t_1$  with the appearance of a `to_be_performed(A)`-atom in the output interface of the agent. No other `tbp`-atoms have occurred before this in the output interface of the agent. The atom `to_be_performed(A)` is transferred by a link to the world, where it arrives some time later, at  $t_2$  (INT3). Our aim now is to prove:

$$\begin{aligned} \text{NoOverlappingInAgent}([0, t_1], \mathcal{M}) & \quad \wedge \\ \text{NoOverlappingInWorld}([0, t_2], \mathcal{M}) & \quad \wedge \\ \text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) & \quad \wedge \\ \text{OccsAndActionsAlternateInWorld}([0, t_2], \mathcal{M}) & \end{aligned}$$

In order to prove the first part of this, we have to show that there are no other `tbp`-atoms occurring in the world before  $t_2$  and that there are no `ended`-atoms before the start of  $A$ , neither in the agent nor in the world. We start with excluding extra `tbp`-atoms in the world. Suppose that at  $t_3 < t_2$  the atom `tbp(B)` arrives in the world. According to INT4, this means that at an earlier time  $t_4 < t_3$  the agent decides to do  $B$ . Now, as `tbp(B)` arrives in the world before `tbp(A)` and we have the no-overtaking property NOOV2, we derive that the agent must have decided to do  $B$  before it decided to do  $A$ . This contradicts with the fact that  $A$  is the first execution starting in the agent. So,  $A$  starting at  $t_2$  is the first action in the world.

Now, we will exclude the occurrence of `ended`-atoms prior to the start of  $A$ . Suppose that at  $t_3 < t_2$  the atom `ended(B)` appears in the output of the world. Property END2 gives us that action  $B$  must have started at an earlier time point  $t_4 < t_3$ . But this yields a contradiction, because we have just proven that no other executions start in the world prior to the start of  $A$  at  $t_2$ . Now, suppose that there is an `ended`-atom arriving in the agent before  $t_1$ . This atom must have come from the world (INT2), which is impossible because we have just shown that the world doesn't produce any `ended`-atoms prior to  $t_2$ .

We now only have to show that occasions and actions alternate until  $t_1$  and  $t_2$ , respectively. This means that each action that starts before  $t_1$  in the agent or  $t_2$  in the world is preceded by its occasion, and no other occasions take place in this time interval. This is trivial, as no actions start prior to  $t_1$  and  $t_2$ .

This finished the proof of the induction basis. We now assume the induction hypothesis:

$$\begin{aligned} \text{IH :} & \\ \text{NoOverlappingInAgent}([0, t_1], \mathcal{M}) & \quad \wedge \\ \text{NoOverlappingInWorld}([0, t_2], \mathcal{M}) & \quad \wedge \\ \text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) & \quad \wedge \\ \text{OccsAndActionsAlternateInWorld}([0, t_2], \mathcal{M}) & \end{aligned}$$

and proceed to the **induction step**. Assuming that the induction hypothesis holds until the start of the  $n$ th execution, we show that we can extend its validity to the beginning of the  $n + 1$ th execution, or (if there is no next execution) to the end of the trace.

We depict the situation in Figure 3.6. We focus on the  $n$ th execution, of action A, which starts at  $t_1$  in the agent and at  $t_2$  in the world. Every start of an execution is followed by its end (END1); suppose the first ended(A) following  $\text{tbp}(A)$  happens at  $t_3$ . This ended-atom arrives in the agent some time later, at  $t_4$  (INT1). In case  $n > 1$ , there is a preceding execution, which is also shown in the figure, and which starts in the agent at  $v_1$  and in the world at  $v_2$ , and ends in the world at  $v_3$  and in the agent at  $v_4$ . In case  $n = 1$ , there is no earlier action execution; we take  $v_1 = v_2 = v_3 = v_4 = 0$ , in order to avoid certain case distinctions in the proof. The start of the (possible)  $n+1$ th execution takes place at  $u_1$  in the agent and  $u_2$  in the world. If the  $n$ th execution was the last one in the trace, we take  $u_1$  and  $u_2$  equal to the time the trace ends, or (if the trace is infinite) to  $\infty$ .

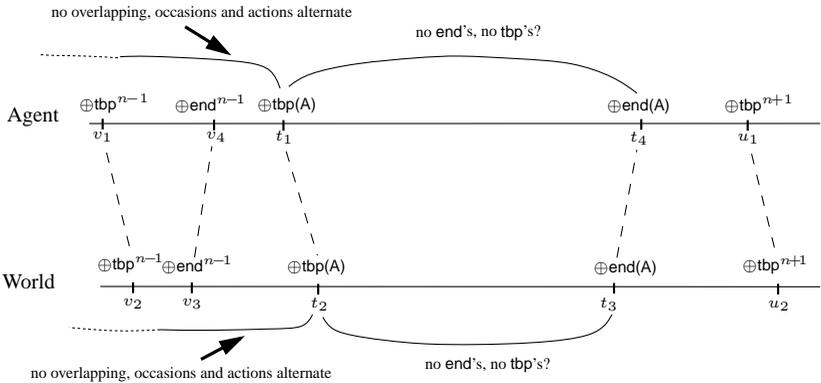


Figure 3.6: Present situation

We now can distinguish six remaining parts to the proof:

- O:** Prove that occasions and actions alternate in the agent and the world, until  $u_1$  and  $u_2$ , respectively.
- I:** Prove that there are no unwanted ended-atoms during the execution of A in the world.
- II:** Prove that there are no unwanted ended-atoms during the execution of A in the agent.

**III:** Prove that there are no unwanted  $\text{tbp}$ -atoms during the execution of  $A$  in the agent.

**IV:** Prove that there are no unwanted  $\text{tbp}$ -atoms during the execution of  $A$  in the world.

**V:** Prove that between the end of the execution of  $A$  and the start of the  $n+1$ th execution no unwanted  $\text{tbp}$ -atoms and  $\text{ended}$ -atoms occur.

We start with part **O**. The IH gives us that

$\text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) \wedge$

$\text{OccsAndActionsAlternateInWorld}([0, t_2], \mathcal{M}),$

and we now have to show that

$\text{OccsAndActionsAlternateInAgent}([0, u_1], \mathcal{M}) \wedge$

$\text{OccsAndActionsAlternateInWorld}([0, u_2], \mathcal{M}).$

At this point in the proof, we will only extend the interval where occasions and actions alternate from  $[0, t_1)$  and  $[0, t_2)$  to  $[0, t_1]$  and  $[0, t_2]$ , respectively. This slight extension of the interval turns out to be the larger part of the proof of part **O**. Later on, we extend the intervals to  $[0, u_1)$  and  $[0, u_2)$  quite effortlessly.

The agent must have had a reason to decide to do  $A$  at  $t_1$  ( $\text{RAT2}$ ); we thus have that there is a  $t_5 < t_1$  such that  $\oplus\text{ObsOcc}(A, t_5, \text{Ag}, \mathcal{M})$ . This information comes from the world;  $\text{INT2}$  gives us that there is a  $t_6 < t_5$  such that  $\oplus\text{ObsOcc}(A, t_6, \text{EW}, \mathcal{M})$ . As observations must have been facts earlier ( $\text{OBS2}$ ), we have a  $t_7 \leq t_6$  such that  $\oplus\text{Occ}(A, t_7, \mathcal{M})$ . Because of the induction hypothesis, we know that  $\text{NoOverlappingInWorld}([0, t_2], \mathcal{M})$ . We use  $\text{ORDER3}$  to conclude that  $\text{NotTooManyOccs}([0, t_2], \mathcal{M})$ . Note that a trace of the whole system  $S$  always contains a trace of  $\text{EW}$ , which is why  $\text{ORDER3}$  is applicable to  $\mathcal{M}$ . We now discern two cases, depending on whether the execution of  $A$  is the first (this is so when  $n = 1$ ) or not.

In case  $n = 1$ , we look at  $A$  being the first action. In this case, we have  $0 \leq t_7 < t_1 < t_2$ . When we look at the meaning of  $\text{NotTooManyOccs}([0, t_2], \mathcal{M})$  as defined in Section 3.5, the first part of the definition states that before the first action in the world, at most one occasion occurs. So, the occurrence of an occasion for  $A$  at  $t_7$  implies that no other occasions happen before  $t_2$ . We conclude that there is exactly one occasion preceding the first action in the world; we have proven  $\text{OccsAndActionsAlternateInWorld}([0, t_2], \mathcal{M})$ .

Now, we will show  $\text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M})$ . This part of the proof is illustrated in Figure 3.7. We already have that the agent observes the occasion for  $A$  at  $t_5 < t_1$ . Suppose the agent observes another occasion for some action  $B$  at time  $t_8 < t_1$ :  $\oplus\text{ObsOcc}(B, t_8, \text{Ag}, \mathcal{M})$ .

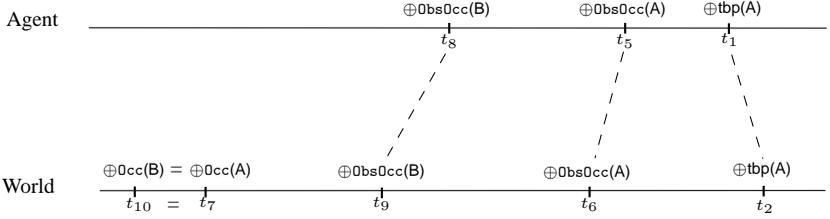


Figure 3.7: Occasions and actions in the agent

In case  $A = B$ , we have that  $t_8 \neq t_5$ , as otherwise there is only one observed occasion (then, both observed occasions coincide). The observation of the occasion for  $B$  comes from the world (INT2), so we have a  $t_9 < t_8$  with  $\oplus\text{ObsOcc}(B, t_9, \text{EW}, \mathcal{M})$ , and in the world there must have been an occasion earlier (OBS2), say at  $t_{10} \leq t_9$ , that gave rise to the observation. As  $t_{10} \leq t_9 < t_8 < t_1 < t_2$ , we now seem to have two occasions occurring before  $t_2$ , one for  $A$  at  $t_7$  and one for  $B$  at  $t_{10}$ . But as we just proved that there is only one occasion preceding the first action in the world, this is impossible. So, we conclude that  $t_{10} = t_7$  and  $A = B$ . As we now have that  $A = B$ , we know that  $t_8 \neq t_5$ , as otherwise the agent observes only one occasion. Now, we have one occasion in the world giving rise to two occasions being observed by the agent. We use the no-overtaking properties to show that this is not possible. First, we use NOOV1. As there are two distinct moments,  $t_8$  and  $t_5$ , that the agent observes an occasion for  $A$ , the second part of NOOV1 gives us that the corresponding moments that the occasions became observable in the output interface of the world are also distinct (as different moments are numbered differently when counted). So,  $t_9 \neq t_6$ . Now, we use the second part of NOOV4, which gives us that when there are distinct (differently numbered) observed occasions in the output interface of the world, the occasions leading to these observations are also (differently numbered, and thus) distinct. We then have  $t_{10} \neq t_7$ , which contradicts with our earlier conclusion that  $t_{10} = t_7$ . Therefore, our initial assumption that the agent observes another occasion for some action  $B$  at time  $t_8 < t_1$  is false, and thus the agent only observes one occasion prior to  $t_1$ , namely the occasion for  $A$  at  $t_5$ . We conclude  $\text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M})$ .

We now come to the second case, which is that  $n > 1$  and  $A$  isn't the first action execution in the system. Recall that we have that at  $t_5 < t_1$  the agent observes the occasion for  $A$ :  $\oplus\text{ObsOcc}(A, t_5, \text{Ag}, \mathcal{M})$ , and that at  $t_7 < t_5$  the occasion happens in the world:  $\oplus\text{Occ}(A, t_7, \mathcal{M})$ . Reactive agents immediately

decide to do an action whenever they observe an occasion. We have the no-overtaking property NOOV3, which implies that two agent decisions must have had two (observed) occasions, where the occasion of the first decision precedes that of the second. So, the agent must have observed the occasion for A later than it observed the occasion for the  $n-1$ th action, as the  $n-1$ th action precedes A. This implies (NOOV1) that the observation of the occasion for A occurred in the output interface of the world later than the observation of the occasion for the  $n-1$ th action did, and thus also that the occasion for A occurring at  $t_7$  in the world takes place after the occasion that led to the  $n-1$ th action execution (NOOV4). As we have  $\text{OCCSAndActionsAlternateInWorld}([0, t_2], \mathcal{M})$  (IH), we know that the occasion for the  $n-1$ th action arose between the start of the  $n-2$ th action (inclusive) and the start of the  $n-1$ th action (exclusive) and that there are no other occasions in this time span. This means that the occasion for A arose at or after the start of the  $n-1$ th action;  $t_7 \in [v_2, t_2]$ . So, there is at least one occasion taking place during the interval  $[v_2, t_2]$ . Now, we need to show  $\oplus \text{OCC}(A, t_7, \mathcal{M})$  is the only occasion happening in this interval. This is easy; it follows from  $\text{NotTooManyOCCS}([0, t_2], \mathcal{M})$ , which we derived before we made the case distinction. The second part of the meaning of this formula allows us to conclude that between the start of the  $n-1$ th execution at  $v_2$  and the start of A at  $t_2$ , only one occasion takes place, namely the occasion for A. We now have shown that  $\text{OCCSAndActionsAlternateInWorld}([0, t_2], \mathcal{M})$ .

Finally, we show that the agent observes only one occasion between  $v_1$  and  $t_1$ . We already have that  $\oplus \text{ObsOCC}(A, t_5, Ag, \mathcal{M})$ , with  $v_1 < v_2 \leq t_7 < t_5 < t_1$ , so we have to show that this is the only one. We reason along the same lines as we did in the case  $n = 1$ , when we showed that the agent observes only one occasion prior to the first action, so we will be brief here. In case there are two observed occasions in the interval  $[v_1, t_1]$ , there would also be two occasions occurring in the world, giving rise to the observed occasions (because of the no-overtaking properties NOOV1 and NOOV4). Because the agent observes both these occasions after it observed the occasion for the  $n-1$ th action, the factual occasions have to take place after the occasion for the  $n-1$ th action. Because we have our earlier conclusion  $\text{OCCSAndActionsAlternateInWorld}([0, t_2], \mathcal{M})$ , we can derive that both of these occasions happen in the interval  $[v_2, t_2]$ , between the start of the  $n-1$ th action and the start of the  $n$ th action, which is impossible because only one occasion is allowed in this interval. So, we have  $\text{OCCSAndActionsAlternateInAgent}([0, t_1], \mathcal{M})$ .

Note that we are not finished with part **O** yet; we need to prove that occasions and actions alternate until  $u_1$  and  $u_2$  respectively, and now we have only reached  $t_1$  and  $t_2$ . We come back to part **O** later on, after we proved parts **I** to **V**, because the rest of part **O** follows easily then.

We proceed to part **I**, which is proving that there are no unwanted ended-atoms during the execution of  $A$  in the world. Suppose there is such an ended-atom disturbing the execution in the world. We want to derive a contradiction in order to exclude this. Suppose that at  $t_5$ , with  $t_2 \leq t_5 \leq t_3$ ,  $\text{ended}(B)$  is derived in the world. As  $\text{ended}(A)$  at  $t_3$  is the first  $\text{ended}(A)$  following the  $\text{tbp}(A)$  at  $t_2$ , we know that  $B \neq A$ . Further suppose that  $t_5$  is the first time that the execution of  $A$  is disturbed by an ended-atom. We discern two cases,  $n = 1$  and  $n > 1$ .

In case  $n = 1$ ,  $A$  is the first action taking place, which is disturbed by  $\text{ended}(B)$  at  $t_5$ . We use **END2** to conclude there is a  $t_6 < t_5$  such that  $\text{tbp}(B)$  arrives in the world.

In case  $n > 1$ , there are other actions preceding  $A$ . We will use **ORDER2** now:

$$\begin{aligned}
& \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A, B \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
& \quad \text{ActionExEnds}(A, t_1, \text{EW}, \mathcal{M}) \quad \wedge \\
& \quad \text{ActionExEnds}(B, t_2, \text{EW}, \mathcal{M}) \quad \wedge \\
& \quad \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) \quad \Rightarrow \\
& \quad \exists t_3 \in \langle t_1, t_2 \rangle : \text{ActionExStarts}(B, t_3, \text{EW}, \mathcal{M})
\end{aligned}$$

Time point  $t_1$  in this property is matched with  $v_3$ , the time the  $n-1$ th execution ends in the world. We substitute  $t_5$  for  $t_2$ . As the IH gives us that  $\text{NoOverlappingInWorld}([0, t_2], \mathcal{M})$ , we also have that  $\text{NoOverlappingInWorld}([0, v_3], \mathcal{M})$ . Then, the property allows us to conclude that there must have been a moment  $t_6$ , with  $v_3 < t_6 < t_5$ , at which an execution of  $B$  starts in the world.

The two cases come together again. Recall that we assume  $v_1 = v_2 = v_3 = v_4 = 0$  when  $n = 1$ . In both cases ( $n = 1$  and  $n > 1$ ) it is impossible that  $t_6 < t_2$ , because we know that action executions take place in an orderly manner (that is, no overlapping) before  $t_2$ , and so if  $B$  starts between  $v_3$  and  $t_2$ , it must end before  $t_2$ , and making the execution of  $A$  the  $n+1$ th execution instead of the  $n$ th. Thus,  $t_2 \leq t_6 < t_5$ . Figure 3.8 illustrates this part of the proof. The  $\text{tbp}(B)$  at  $t_6$  comes from the agent (**INT4**). Because there is no overlapping of action executions before  $t_1$  (IH) and thus there are no actions starting between  $v_4$  and  $t_1$ , the agent must have derived  $\text{tbp}(B)$  at  $t_7$ , with  $t_1 \leq t_7$ . Because  $t_7 < t_6 < t_5 \leq t_3 < t_4$ , we know  $t_7 < t_4$ . So, we now have a disturbing  $\text{tbp}$ -atom in the agent at  $t_7$ . **ORDER1** is a property about disturbing  $\text{tbp}$ -atoms:

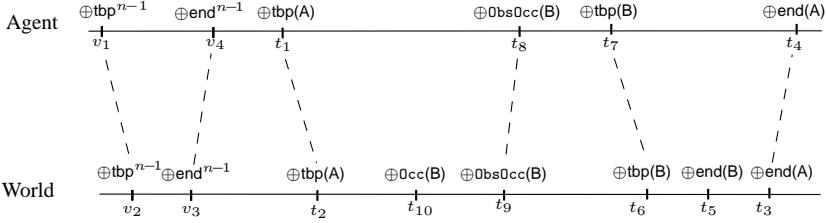


Figure 3.8: Excluding ended-atoms during A in the world

$$\begin{aligned}
& \forall \mathcal{M} \in \text{Traces}(\text{Ag}) \quad \forall A, B \in \text{ACTION} \quad \forall t_1 \quad \forall t_2 \geq t_1 : \\
& \quad \text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) \quad \quad \quad \wedge \\
& \quad \text{ActionExStarts}(B, t_2, \text{Ag}, \mathcal{M}) \quad \quad \quad \wedge \\
& \quad \text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) \quad \quad \Rightarrow \\
& \exists t_3 \in [t_1, t_2] : \oplus \text{ObsOcc}(B, t_3, \text{Ag}, \mathcal{M})
\end{aligned}$$

We match  $t_1$  in this property with  $t_1$  in the figure, and we match  $t_2$  in the property with  $t_7$  in the figure. We proved

$\text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M})$  in part **O**, so we are rightfully concluding that there is a  $t_8$  with  $t_1 \leq t_8 < t_7$  such that the agent observes an occasion for B at  $t_8$ . Note that this already excludes that  $t_7 = t_1$ . Observations come from the world (INT2), so there must have been a  $t_9 < t_8$  such that  $\oplus \text{ObsOcc}(B, t_9, \text{EW}, \mathcal{M})$ , and observations were facts (OBS2), so there is a  $t_{10} \leq t_9$  with  $\oplus \text{Occ}(B, t_{10}, \mathcal{M})$ . Because in part **O** we proved that  $\text{OccsAndActionsAlternateInWorld}([0, t_2], \mathcal{M})$  it can't be that  $t_{10} < t_2$ . We thus have  $t_2 \leq t_{10} \leq t_9 < t_8 < t_7 < t_6 < t_5 \leq t_3$ , so  $t_2 \leq t_{10} < t_3$ . As there is an occasion occurring at  $t_{10}$ , this must be the effect of either an event or an action (CHANGE). Now, we use ORDER4:

$$\begin{aligned}
& \forall \mathcal{M} \in \text{Traces}(\text{EW}) \quad \forall A \in \text{ACTION} \quad \forall t_1 \quad \forall t_2 > t_1 : \\
& \quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \quad \quad \wedge \\
& \quad \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) \quad \quad \quad \Rightarrow \\
& \quad \text{NoEventsDuring}([t_1, t_2], \mathcal{M})
\end{aligned}$$

We substitute  $t_2$  for  $t_1$  and  $t_3$  for  $t_2$  in the property. The NoOverlapping condition is fulfilled because of the IH. Thus, we conclude that no events happen in  $[t_2, t_3]$ . Consequently, the occasion at  $t_{10}$  must be an action effect. Now we use ORDER5:

$$\begin{aligned}
& \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A, B \in \text{ACTION} \forall t_1 \forall t_2 > t_1 \forall t_3 \in [t_1, t_2) : \\
& \quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) && \wedge \\
& \quad \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) && \wedge \\
& \quad \text{OccursAndActionsAlternateInWorld}([0, t_1], \mathcal{M}) && \wedge \\
& \quad \text{NoEventsDuring}([t_1, t_2], \mathcal{M}) && \wedge \\
& \quad \oplus \text{obsOcc}(B, t_3, \text{EW}, \mathcal{M}) && \Rightarrow \\
& \exists C \in \text{ACTION} : \text{ActionExEnds}(C, t_3, \text{EW}, \mathcal{M})
\end{aligned}$$

Again, we substitute  $t_2$  for  $t_1$  and  $t_3$  for  $t_2$ . The `NoOverlapping` condition is fulfilled, and we have just concluded that `NoEventsDuring` $([t_2, t_3], \mathcal{M})$ . We substitute  $t_9$  for  $t_3$ , and conclude that there is an action  $C$  such that `ActionExEnds` $(C, t_9, \text{EW}, \mathcal{M})$ . But  $t_9 < t_5$ , which contradicts our initial assumption that  $t_5$  was the first time that the execution of  $A$  was disturbed by an ended-atom. We finish part **I** by concluding that no other actions end during the execution of  $A$  in the world.

Now, we arrive at part **II**, proving that there are no unwanted ended-atoms during the execution of  $A$  in the agent. Suppose that this is not true, so suppose there is a  $t_5$  with  $t_1 \leq t_5 \leq t_4$  such that there arrives a disturbing ended( $B$ ) in the agent. In case  $t_5 = t_4$ , we have  $B \neq A$ . This ended( $B$ ) comes from the world (INT2), so there must have been a  $t_6 < t_5$  such that  $B$  ends in the world. The situation is depicted in the following figure:

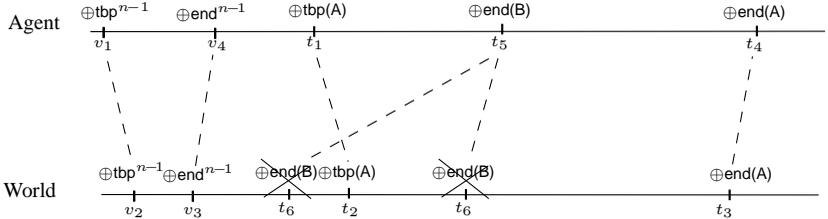


Figure 3.9: Excluding ended-atoms during  $A$  in the agent

In the agent, ended( $B$ ) happens between the ended-atom designating the end of the  $n-1$ th action execution and ended( $A$ ). So, because of the no-overtaking property NOOV1 this must also be the case in the world. There are two possibilities for  $t_6$ :  $t_6$  lies between the end of the  $n-1$ th action execution and  $t_2$  ( $v_3 < t_6 < t_2$ ) or  $t_6$  is during the execution of  $A$  ( $t_2 \leq t_6 \leq t_3$ ). The first possibility is impossible because the IH gives us `NoOverlappingInWorld` $([0, t_2], \mathcal{M})$ , and thus no `tpb`-atoms or ended-atoms occur between  $v_3$  and  $t_2$ , and the sec-

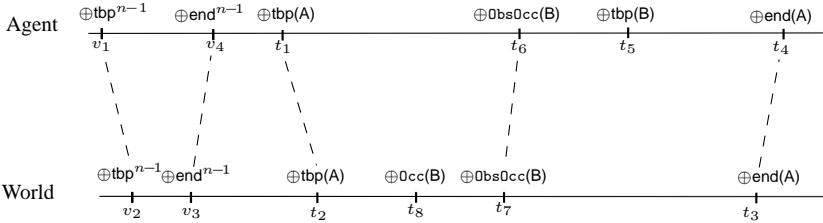


Figure 3.10: Excluding  $\text{tbp}$ -atoms during  $A$  in the agent

ond possibility is excluded because we just proved, in part **I**, that there are no ended-atoms disturbing the  $n+1$ th execution in the world.

Part **III** is next. Here, we have to prove there are no unwanted  $\text{tbp}$ -atoms during the execution of  $A$  in the agent. Suppose there is a disturbing  $\text{tbp}(B)$ -atom at  $t_5$  with  $t_1 \leq t_5 \leq t_4$ , with  $B \neq A$  in case  $t_5 = t_1$ . As the next part of the proof is analogous to the last part of the proof of part **I**, we will proceed somewhat sloppily. We picture this part of the proof in Figure 3.10. We use **ORDER1** to conclude that there is a  $t_6$  with  $t_1 \leq t_6 < t_5$  such that  $\oplus\text{obsOcc}(B, t_6, \text{Ag}, \mathcal{M})$ . This observation comes from the world: at a  $t_7 < t_6$  we have  $\oplus\text{obsOcc}(B, t_7, \text{EW}, \mathcal{M})$  and at a  $t_8 \leq t_7$  we have  $\oplus\text{Occ}(B, t_8, \mathcal{M})$ . Because occasions and actions alternate up to  $t_2$ , we have  $t_8 \geq t_2$ . Because there are no events during the execution of  $A$  in the world (**ORDER4**), we can use **ORDER5** to conclude that there must be an action ending at  $t_7$ , and thus also in the agent at  $t_6$ . As  $t_1 < t_2 \leq t_8 \leq t_7 < t_6 < t_5 \leq t_4$ , we now have an ended-atom disturbing an execution in the agent, which we proved impossible in part **II**.

We now are at part **IV**, in which we show there are no unwanted  $\text{tbp}$ -atoms during the execution of  $A$  in the world. Suppose there is a disturbing  $\text{tbp}(B)$ -

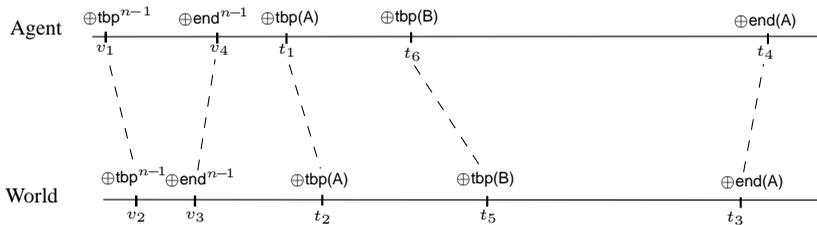


Figure 3.11: Excluding  $\text{tbp}$ -atoms during  $A$  in the world

atom occurring at  $t_5$ , with  $t_2 \leq t_5 \leq t_3$  and  $B \neq A$  in case  $t_5 = t_2$ . The situation is depicted in Figure 3.11. The  $\text{tbp}(B)$ -atom must have come from the agent (INT4), so  $\text{tbp}(B)$  became *true* at  $t_6$  with  $t_6 < t_5$ . The IH gives us that  $\text{NoOverlappingInAgent}([0, t_1], \mathcal{M})$ , and so no  $\text{tbp}$ -atoms or ended-atoms occur between  $v_4$  and  $t_1$ . Thus, we know  $t_6 \geq t_1$ . Furthermore, we know  $t_6 < t_5 \leq t_3 < t_4$ , so  $t_6 < t_4$ . But then,  $\text{tbp}(B)$  happens during the  $n$ th execution in the agent, and in part **III** we proved this to be impossible.

So, we have showed that the  $n$ th execution is not disturbed by anything; we have

$$\begin{aligned} & \text{NoOverlappingInWorld}([0, t_3], \mathcal{M}) \quad \wedge \\ & \text{NoOverlappingInAgent}([0, t_4], \mathcal{M}) \end{aligned}$$

We proceed with part **V**, in which we prove that between the end of the execution of  $A$  and the start of the  $n+1$ th execution no unwanted  $\text{tbp}$ -atoms and ended-atoms occur. In other words, we extend the intervals where there is no overlapping of action executions to  $[0, u_2)$  and  $[0, u_1)$ , respectively. In case there is no  $n+1$ th action execution, we extend the interval up to the end of the trace.

So, at time  $u_1 > t_4$  the  $n+1$ th action starts in the agent. No other  $\text{tbp}$ -atoms occur in the agent in the interval  $\langle t_4, u_1 \rangle$ . The  $\text{tbp}$ -atom is transferred to the world and it arrives there at  $u_2 > u_1$  (INT3). We now have to show that:

- A:** There are no other  $\text{tbp}$ -atoms occurring in the world between  $t_3$  and  $u_2$ .
- B:** There are no ended-atoms occurring in the world between  $t_3$  and  $u_2$ .
- C:** There are no ended-atoms occurring in the agent between  $t_4$  and  $u_1$ .

We start with **A**. In Figure 3.12, we depict the situation. Some aspects of the picture relate to proof steps we will make soon.

Suppose there is a  $\text{tbp}$ -atom (say,  $\text{tbp}(B)$ ) becoming *true* in the world during  $\langle t_3, u_2 \rangle$ , then this atom must have become *true* some time earlier in the agent (INT4), but after the  $n$ th  $\text{tbp}$ -atom (which is  $\text{tbp}(A)$ ) was derived (NOOV2). Two potentially possible moments are shown in Figure 3.12. Because the  $n$ th action execution is living up to  $\text{COORD0}$ , as we proved in the part **I** to **IV**, it is impossible that  $\text{tbp}(B)$  happens during the  $n$ th execution. And  $\text{tbp}(B)$  cannot have happened after the end of the  $n$ th execution but before  $u_1$ , because we posited that the  $\text{tbp}$ -atom at  $u_1$  is the first action start following  $t_4$ . So, we conclude that there are no  $\text{tbp}$ -atoms arriving during  $\langle t_3, u_2 \rangle$  and the execution starting at  $u_2$  in the world is the first action beginning since  $t_2$ .

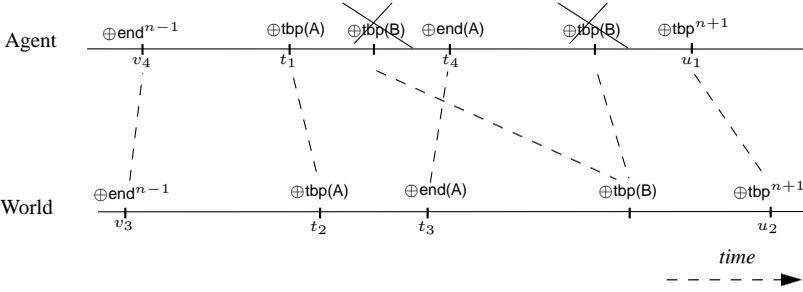


Figure 3.12: Excluding actions starting between the  $n$ th and  $n+1$ th execution

**Part B** is next. Suppose that action **B** ends in the world at time  $t_5 \in \langle t_3, u_2 \rangle$ . We now have two consecutive endings of actions, namely **A** at time  $t_3$  and **B** at  $t_5$ . We use **ORDER2**, which states:

$$\begin{aligned}
& \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A, B \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
& \quad \text{ActionExEnds}(A, t_1, \text{EW}, \mathcal{M}) \quad \wedge \\
& \quad \text{ActionExEnds}(B, t_2, \text{EW}, \mathcal{M}) \quad \wedge \\
& \quad \text{NoOverlappingInWorld}([0, t_1], \mathcal{M}) \quad \Rightarrow \\
& \exists t_3 \in \langle t_1, t_2 \rangle : \text{ActionExStarts}(B, t_3, \text{EW}, \mathcal{M})
\end{aligned}$$

We instantiate  $t_1$  from the property with  $t_3$  and  $t_2$  from **ORDER2** with  $t_5$ . The **NoOverlapping** condition is fulfilled, as we concluded after finishing part **IV**. Then, the conclusion of **ORDER2** yields that there must have been an execution of **B** starting at some time between  $t_3$  and  $t_5$ . This contradicts the fact that no **tbp**-atoms arrive in the world between  $t_3$  and  $u_2$  which we proved at **A**.

We turn to part **C**, which is showing that there are no ended-atoms arriving in the agent during  $\langle t_4, u_1 \rangle$ . Suppose **ended(B)** becomes *true* in the input of the agent at  $t_5 \in \langle t_4, u_1 \rangle$ . This atom comes from the world (**INT2**), where **ended(B)** occurred at a moment  $t_6 < t_5 < u_1 < u_2$ . As **ended(B)** follows **ended(A)** at  $t_4$ , the no-overtaking property **NOOV1** gives us that  $t_6 > t_3$ . But now we have an action ending at  $t_6 \in \langle t_3, u_2 \rangle$ , which we proved impossible in **B**.

As no unwanted **tbp**-atoms or ended-atoms occur between the end of the  $n$ th execution and the beginning of the  $n+1$ th execution, we have shown that there is no overlapping of action executions until the start of the  $n+1$ th execution. We have finished part **V**, and we now established:

$$\begin{aligned}
& \text{NoOverlappingInAgent}([0, u_1], \mathcal{M}) \quad \wedge \\
& \text{NoOverlappingInWorld}([0, u_2], \mathcal{M})
\end{aligned}$$

To close the proof, we return to part **O**. We already proved

$$\begin{aligned} & \text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) \quad \wedge \\ & \text{OccsAndActionsAlternateInWorld}([0, t_2], \mathcal{M}) \end{aligned}$$

We can simply extend these properties until  $u_1$  and  $u_2$ , respectively, because no other action executions start until these moments in the agent or the world, respectively. This means that there are no new pairs of consecutive actions starting in the prolonged interval  $[0, u_1)$  and  $[0, u_2)$ , and that we thus have considered all consecutive action beginnings in this interval in our earlier account of part **O**. We now have proven:

$$\begin{aligned} & \text{NoOverlappingInAgent}([0, u_1), \mathcal{M}) \quad \wedge \\ & \text{NoOverlappingInWorld}([0, u_2), \mathcal{M}) \quad \wedge \\ & \text{OccsAndActionsAlternateInAgent}([0, u_1), \mathcal{M}) \quad \wedge \\ & \text{OccsAndActionsAlternateInWorld}([0, u_2), \mathcal{M}) \end{aligned}$$

which is the induction hypothesis holding until the start of the  $n+1$ th action execution (or the end of the trace). Induction then gives us that the above properties hold indefinitely, which implies that **COORD0** is valid.

### 3.8 Proving **COORD5**

**COORD5** states that whenever an execution of an action starts in the world, the action is applicable. This is a valuable property, as actions will not yield the desired effects if they're not applicable. We repeat **COORD5** here:

$$\begin{aligned} & \text{COORD5 :} \\ & \forall \mathcal{M} \in \text{Traces}(\mathbf{S}) \quad \forall A \in \text{ACTION} \quad \forall t_1 \quad \forall t_2 > t_1 : \\ & \quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \Rightarrow \\ & \quad \text{Appl}(A, t_1, \mathcal{M}) \end{aligned}$$

Like **COORD0**, **COORD5** is a system property that only holds when both the agent and the world behave properly. Again, the details of proving **COORD5** depend on the nature on the agent. In this chapter, as stated earlier, we focus at reactive agents.

Before we present the properties, we give a first sketch of the proof. When an action starts in the world, as the condition of **COORD5** states, this is caused by the agent earlier observing an occasion. As observations always are true,

we know that the action must have been occasional, and thus applicable, earlier on. We demand that neither events nor action effects can take place that disturb this applicability, and thus the action is still applicable at the time the execution starts. Perhaps surprisingly, we only need demands on the world behaviour to prove that neither events nor action effects disrupt applicability. The reason for this lies in the reactive nature of the agent; the agent can't help deciding that it will do an action when it sees an occasion. If we want to prevent the agent from disrupting applicability of an action A by doing other interfering actions before A starts, then we have to constrain the generation of occasions in the world, such that the agent won't decide to do these disturbing actions.

The ORDER properties we introduced in the previous section already constrain the dynamics of the world considerably. In the first subsection, we will present two additional properties of the world which we need for proving COORD5. As always with properties of the world (and of the agent), we won't try to prove these properties from yet other properties, but regard them as demands on the dynamics of the world. In the second subsection, we present the proof of COORD5.

### 3.8.1 Properties needed for proving COORD5

The additional properties we need for proving COORD5 are named PROPER1 and PROPER2, as they are needed to prove that the state of the world will be proper for action execution whenever an action starts.

In the world, events can take place that change the state of the world, cancelling applicability of actions. We will forbid events like this taking place during critical time spans in property PROPER1. Informally, this property states:

'When an action execution starts in the world at  $t_1$   
and  
action executions don't overlap in the world  
and  
the most recent time that an occasion for the action arose is at  $t_2$   
and  
the most recent time that the action became applicable is at  $t_3$   
and  
at some moment after  $t_2$  and  $t_3$  but before  $t_1$  the action becomes  
not applicable  
then  
there were no events at that moment.'

Both at  $t_2$  and  $t_3$ , the action is applicable, because occasionality implies applicability. We need to take into account both these moments in order to keep the time frame where we forbid events as small as possible. For the same reason, we only forbid events at those times where applicability is indeed disturbed. This is the formalisation of PROPER1 where  $\max(t_1, t_2)$  is the maximum of  $t_1$  and  $t_2$ :

$$\begin{array}{l}
 \text{PROPER1 :} \\
 \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 < t_1 \\
 \forall t_3 < t_1 \forall t \in \langle \max(t_2, t_3), t_1 \rangle : \\
 \quad \text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M}) \quad \wedge \\
 \quad \text{NoOverlappingInWorld} \quad \wedge \\
 \quad - \otimes t_1, t_2 \oplus \text{Occ}(A, t_2, \mathcal{M}) \quad \wedge \\
 \quad - \otimes t_1, t_3 \oplus \text{Appl}(A, t_3, \mathcal{M}) \quad \wedge \\
 \quad \oplus \neg \text{Appl}(A, t, \mathcal{M}) \quad \Rightarrow \\
 \quad \text{NoEventsDuring}([t, t], \mathcal{M})
 \end{array}$$

This property stipulates that when action A suddenly becomes not applicable, this can't be due to events, as these never happen at these moments.

We need another world property, that formalises that all effects of an action happen at the same time. The reason for introducing this property is that when an action has multiple effects at different moments during the execution, a later effect can nullify an earlier effect. And if the earlier effect caused an occasion for a new action, then the agent will decide to do the new action, which can turn out to be not applicable, if the later effect creates a world state in which the new action is not properly executable.

Informally, this is PROPER2:

‘When there is an action execution in the world  
and  
action executions don't overlap in the world  
and  
there are two action effects during this execution  
then  
they happen simultaneously.’

And this is the formal form:

PROPER2

$$\begin{array}{l}
\forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall l, k \in \text{groundliterals}(\text{world info}) \\
\forall t_1 \forall t_2 > t_1 \forall t_3, t_4 \in \langle t_1, t_2 \rangle : \\
\quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \wedge \\
\quad \text{NoOverlappingInWorld} \quad \wedge \\
\quad \text{ActionEff}(A, l, t_3, \mathcal{M}) \quad \wedge \\
\quad \text{ActionEff}(A, k, t_4, \mathcal{M}) \quad \Rightarrow \\
\quad t_3 = t_4
\end{array}$$

### 3.8.2 The proof of COORD5

The following proof is illustrated in Figure 3.13. We have to prove:

$$\begin{array}{l}
\forall \mathcal{M} \in \text{Traces}(\text{S}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\
\quad \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \Rightarrow \\
\quad \text{App1}(A, t_1, \mathcal{M})
\end{array}$$

In order to do this, we take an arbitrary trace  $\mathcal{M}$  of  $\text{S}$ , an action  $A$  and time points  $t_1, t_2$  such that  $\text{ActionEx}(A, t_1, t_2, A, \mathcal{M})$ . This implies that  $\oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{EW})) \models \text{to\_be\_performed}(A)$ , which is also written as  $\text{ActionExStarts}(A, t_1, \text{EW}, \mathcal{M})$ . By INT4, we know that  $\text{to\_be\_performed}(A)$  became *true* in the output interface of the agent earlier, at  $t_3 < t_1$ ; in case there are more candidates for  $t_3$ , we take the latest moment. RAT2 gives us that the agent must have observed an occasion some time earlier, at  $t_4 < t_3$ , where we take the latest  $t_4$  possible in case there are several observed occasions for  $A$  prior to  $t_3$ . This observation comes from the world; by INT2, we deduce that at some earlier time point there was an occasion for  $A$  observed in the world. By OBS2, the observation must correspond to an earlier fact. So, we have that at  $t_5 < t_4$ ,  $\oplus \text{occ}(A, t_5, \mathcal{M})$ . We again choose  $t_5$  as late as possible. Figure 3.13 shows the situation. Because occasionality implies applicability (RAT3), we have  $\text{App1}(A, t_5, \mathcal{M})$ .

Changes can take place in the world situation between  $t_5$  and  $t_1$ , making  $A$  no longer applicable. These changes will do no harm when later on, but earlier than  $t_1$ , there are again changes that make  $A$  applicable again. In case disrupting changes occur, they shouldn't happen anymore after the last time the action becomes applicable before  $t_1$ . Say we have  $t_6 < t_1$  with

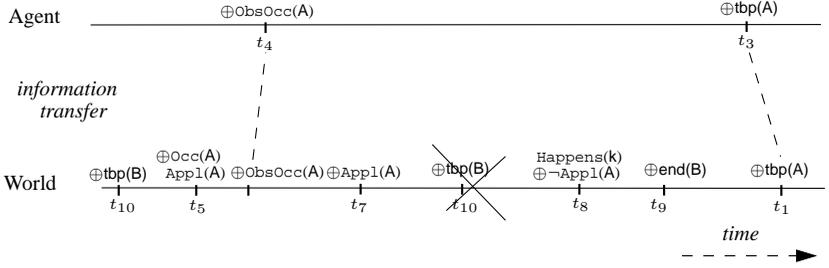


Figure 3.13: Sketch of the situation

–  $\otimes t_1, t_6 \oplus \text{App1}(A, t_6, \mathcal{M})$ . If no changes affecting applicability take place after  $t_5$ , it could be that  $t_6$  is earlier than  $t_5$ . In this case, we want to exclude disrupting changes from  $t_5$ , as we know that A was applicable at  $t_5$ . We take  $t_7 = \max(t_5, t_6)$ ; we have  $t_5 \leq t_7 < t_1$ , and  $\text{App1}(A, t_7, \mathcal{M})$ .

In the induction proof of `COORD0` we have also proven that `OccursAndActionsAlternateInWorld`( $[0, t], \mathcal{M}$ ), for each trace  $\mathcal{M}$  and time point  $t$ . As a consequence of this, and of the way we chose  $t_3, t_4$  and  $t_5$  to be as close to  $t_1$  as possible, no other actions can start between  $t_5$  and  $t_1$ , and no other occasions can arise in this period.

Now, suppose there is a moment  $t_8 \in \langle t_7, t_1 \rangle$  such that  $\neg \text{appl}(A)$  becomes *true*. At least one ground literal of world info must have changed truth value to *true* then. Take one such literal and call it  $k$ . We thus have `Happens`( $k, t_8, \mathcal{M}$ ). Now, we use `CHANGE` to conclude that  $k$  either is an action effect or an event effect. We thus have two cases.

The second case, that  $k$  is an event effect, isn't possible, as easily follows from `PROPER1`. We instantiate  $t_1$  of `PROPER1` with our present  $t_1$ ,  $t_2$  with  $t_5$  and  $t_3$  with  $t_6$ . Then, the interval  $\langle \max(t_2, t_3), t_1 \rangle$  is interval  $\langle t_7, t_1 \rangle$  in the figure. We instantiate  $t$  with  $t_8$ . The `NoOverlapping-condition` is fulfilled because we have `COORD0`, and the other conditions are fulfilled by the way we instantiated the variables. We then conclude `NoEventsDuring`( $[t_8, t_8], \mathcal{M}$ ), which means that  $k$  can't be an `EventEffect`.

We now look at the first case, which is that  $k$  is an action effect. We have  $\exists B \in \text{ACTION} : \text{ActionEff}(A, k, t_8, \mathcal{M})$ . This means that  $k$  is enclosed within a `tbp(B)–end(B)` pair, with `ended(B)` becoming *true* at  $t_9 \geq t_8$ . Action B must have started before  $t_8$ . In the figure, we show two options for this moment, namely  $t_{10} > t_5$  and  $t_{10} \leq t_5$ . But it is impossible that B begins after  $t_5$ , as earlier we showed that no other actions start between  $t_5$  and  $t_1$ . So, B begins before or at the same time the occasion for A arises. Now, `ORDER4`,

from the previous section, is used:

$$\begin{aligned} \forall \mathcal{M} \in \text{Traces}(\text{EW}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 : \\ \text{ActionEx}(A, t_1, t_2, \text{EW}, \mathcal{M}) \quad \wedge \\ \text{NoOverlappingInWorld}([0, t_1), \mathcal{M}) \quad \Rightarrow \\ \text{NoEventsDuring}([t_1, t_2), \mathcal{M}) \end{aligned}$$

We substitute B for A in the property,  $t_{10}$  for  $t_1$  and  $t_9$  for  $t_2$ . The `NoOverlapping` condition is fulfilled because `COORD0` is valid. Then, we can conclude that no events take place during the execution of B. But then the occasion for A arising at  $t_5$  must be an effect of the action B! Using `PROPER2`, that states that two action effect of one action happen at the same time, we can conclude that  $t_8 = t_5$ . As  $t_8 \in (t_7, t_1]$  and  $t_7 \geq t_5$ , we can also conclude  $t_8 > t_5$ , and we have a contradiction. This means that k can't be an action effect either.

As both cases are impossible, our initial assumption that there was a moment  $t_8 \in (t_7, t_1]$  such that  $\oplus \neg \text{App1}(A, t_8, \mathcal{M})$ , was false. This means that  $\text{appl}(A)$  continues to hold and still holds at  $t_1$ :  $\text{App1}(A, t_1, \mathcal{M})$ .

### 3.9 Verifying an instance of the class: the Mouse

This section is a bird's-eye view on applying the system of coordination properties to a specific system. Now that we presented the generic system of coordination properties and the proof of action successfulness using these properties, we will sketch how these generic results can be used to guarantee action successfulness for a concrete instantiation of the generic architecture. For the concrete system, we turn to our running example of the artificial mouse. In order to use the system of coordination properties, we need to fill in the system-dependent details. Next, the agent- and world properties of the system of coordination properties must be shown to hold for the mouse system. The generic proofs (of `COORD0`, `COORD5` and `COORD1`) don't need any adjustment; they are valid proofs for each instantiation of the generic architecture (provided that the agent is reactive, for the proof of `COORD0` and `COORD5`). When indeed all properties of the agent and the world (and the interaction between them) mentioned in this chapter hold, then action successfulness holds for the mouse system.

The first thing that needs to be done when using the generic properties and proofs is instantiating the domain-dependent features. The generic architecture as described in Section 3.2 has several ontologies which are (partly) domain-dependent. The ontology world info is left totally unspecified in the generic architecture, and when verifying the mouse system, we need to instantiate this

ontology with the information structures used in the mouse system. We already explained in Section 3.2 that world info for the mouse system provides ground atoms of the form  $\text{at\_position}(O, P)$ , where  $O$  is one of mouse, food and screen and  $P$  is one of p0, p1 and p2. The ontologies actions to be performed and actions finished both make use of the sort ACTION, which is domain dependent and needs to be instantiated. In case of the mouse system, ACTION is the set  $\{\text{goto}(p0), \text{goto}(p1), \text{goto}(p2), \text{eat}\}$ .

Not only ontologies are domain-dependent. Several notions used in the properties also need to be provided for each specific instantiation of the generic architecture. We presented these notions scattered throughout Section 3.5, and here we repeat them:

- \*  $\text{appl}(A)$ , for  $A \in \text{ACTION}$ , is a formula describing in which world situations  $A$  is applicable.
- \*  $\text{occ}(A)$ , for  $A \in \text{ACTION}$ , is a formula describing when there is an occasion for  $A$ .
- \*  $\text{sit\_spec\_effect}$  is a ternary relation formalising what the effects of a certain action in a certain world situation will be.

We gave examples of instantiations of these notions for the mouse system in Section 3.5.

Secondly, we have to anchor the system of coordination properties. The proofs we did in previous sections generally concerned system properties, that were proven from agent properties and world properties. We need to be sure that these properties hold for the particular agent and world present in the system, otherwise the coordination properties are not guaranteed to be true. But we didn't prove these underlying agent- and world properties, because the proofs of these depend on the particular implementation of the agent and on the nature of the world. For example, the world could be a real physical environment, which isn't controlled by the system in any way. In this case, the world properties are merely assumptions on the world behaviour, under which proper system behaviour is obtained. But the world can also be a simulated environment, which is programmed in some language. In this case, the world properties should be proven from the semantics of the specification of the world. The same holds for the agent; in order to prove the agent properties, we need to have an implementation or specification of the agent in some programming language or specification language with a well-defined semantics. Having this, we can try to prove the agent properties.

At this point, there are many difficulties. Most agent programming languages and specification frameworks don't have a well-defined formal seman-

tics. In the next chapters of this thesis, we will try to address this problem by designing a number of agent programming languages with well-defined semantics. Other people who have worked on agent specification languages with a firm formal semantics are David Kinny [80], Pascal van Eck [16, 39], Koen Hindriks [59] and Rogier van Eijk [12].

We again look at the implementation of the agent in the mouse system. This is the decision model of the mouse, implementing the reactive nature of its behaviour:

```

if    observation_result(at_position(food, P) ∧
                          ¬at_position(mouse, P) ∧
                          ¬at_position(screen, p0)    )
then  to_be_performed(goto(P))

if    observation_result(at_position(mouse, P) ∧
                          at_position(food, P)        )
then  to_be_performed(eat)

```

Though we now have the knowledge base of the agent in the mouse system, we can't prove the agent properties of the system of coordination properties from this alone. This is because we need information regarding the control cycle of the agent. We don't know whether the knowledge base is continuously reasoning, while new observation results come in at all times, or whether it only reasons during specific time intervals, possibly closing down to new observation results. This depends on the particular framework or language used to build the knowledge base, and its unambiguous formal semantics. We choose not to go into details of this in this chapter, because it is already quite lengthy. The rules of the knowledge base are intuitive enough to give us an idea of how the agent works, but as we don't look at the precise temporal semantics of the rules, this is not enough to prove the agent properties.

Therefore, we will only give intuitive justifications of the agent properties mentioned in this chapter. We only want to give an idea of how to use the generic properties and proofs. We won't go into world properties and properties of interaction between agent and world.

In the system of coordination properties and among the basic properties (Section 3.6), there are only four agent properties. We start with the two agent rationality properties that are phrased in Section 3.6.2:

$$\begin{aligned}
 \text{RAT1 :} \\
 \forall \mathcal{M} \in \text{Traces}(\text{Ag}) \forall A \in \text{ACTION} \forall t_1 : \\
 \quad \oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\
 \exists t_2 > t_1 : \text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M})
 \end{aligned}$$

$$\begin{aligned}
& \text{RAT2 :} \\
& \forall \mathcal{M} \in \text{Traces}(\text{Ag}) \forall A \in \text{ACTION} \forall t_2 : \\
& \quad \text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M}) \Rightarrow \\
& \quad \exists t_1 < t_2 : \oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M})
\end{aligned}$$

According to the rules of the mouse, it will decide to start an action when it has an observation of an occasion for that action. As reasoning takes some time, the properties above are reasonable.

Next, there is an agent property used in proving COORD0:

$$\begin{aligned}
& \text{ORDER1 :} \\
& \forall \mathcal{M} \in \text{Traces}(\text{Ag}) \forall A, B \in \text{ACTION} \forall t_1 \forall t_2 \geq t_1 : \\
& \quad \text{ActionExStarts}(A, t_1, \text{Ag}, \mathcal{M}) \quad \wedge \\
& \quad \text{ActionExStarts}(B, t_2, \text{Ag}, \mathcal{M}) \quad \wedge \\
& \quad \text{OccsAndActionsAlternateInAgent}([0, t_1], \mathcal{M}) \quad \Rightarrow \\
& \quad \exists t_3 \in [t_1, t_2) : \oplus \text{ObsOcc}(B, t_3, \text{Ag}, \mathcal{M})
\end{aligned}$$

Suppose the conditions of this property hold for a certain agent trace and certain actions A and B and moments  $t_1$  and  $t_2 \geq t_1$ . Then we know that B starts after A, and that up to the beginning of A, observed occasions and action beginnings alternate. Now, the mouse must also have observed an occasion for B. This occasion must be distinct from all earlier observed occasions (which give rise to all earlier actions), as action B follows all earlier actions. This means that the observed occasion for B must arrive in the agent after the observed occasions of the earlier actions, including A. As there can be only one observed occasion prior to each action, the observed occasion for B must take place after action A has started and before B begins. This makes ORDER1 an acceptable property.

Finally, we have the no-overtaking property for the information processing of the agent:

$$\begin{aligned}
& \text{NOOV3 :} \\
& \forall \mathcal{M} \in \text{Traces}(\text{Ag}) \forall A \in \text{ACTION} \forall t_1 \forall t_2 > t_1 \forall n \in \mathbb{N} \setminus \{0\} \\
& \quad [\oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ObsOccCount}(n, t_1, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ActionCount}(n, t_2, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\
& \quad \text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M})] \quad \wedge \\
& \quad [\text{ActionExStarts}(A, t_2, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ActionCount}(n, t_2, \text{Ag}, \mathcal{M}) \wedge \\
& \quad \text{ObsOccCount}(n, t_1, \text{Ag}, \mathcal{M}) \quad \Rightarrow \\
& \quad \oplus \text{ObsOcc}(A, t_1, \text{Ag}, \mathcal{M})]
\end{aligned}$$

This property demands that the reasoning of the knowledge base of the mouse is implemented in such a way that the mouse generates separate decisions (in time) to do actions when the mouse observes the occasions for the actions at separate moments. This is a restriction on the implementation of the control on the reasoning of the mouse. The knowledge base must be used in such a way to process incoming information into conclusions that the above property holds.

It might be striking that the majority of properties used for proving action successfulness for reactive single agents in dynamic environments are demands on the world, and that we only have four agent properties. The reason for this is that reactive agents are not very intelligent, and don't have means to adjust their decisions to environmental circumstances over time. When the agent isn't flexible, the demands on the world behaviour become heavier, as this chapter clearly demonstrates.

### 3.10 Discussion

One of the challenges of improving development methods for agent systems is to provide appropriate support for verification of agent systems being built in practice. The current state of affairs is that from the theoretical side formal techniques are proposed, such as temporal logics, but that developers in practice do not consider them useful. Three main reasons for this gap are that

- \* behavioural properties relevant for agent systems in practice usually have such a high complexity that both fully automated verification and verification by hand are difficult,
- \* the formal techniques offered have no well-defined relation to design or software specifications of the real systems used in practice, and
- \* the formal techniques offered require a much higher level of formal skills than the average developer in practice possesses.

In this chapter we address these issues in the following manner. Two languages are proposed: a detailed language, with a direct relation to the system design specification, and an abstract language in which properties can be formulated in a more conceptual manner. Both languages have been defined formally; moreover, well-defined relationships exist between the two languages (Section 3.5). The relation between the detailed language and the system specification language is also well-defined, as the detailed language is built from the logical constructs used in the system specification language, as explained in Section 3.3.1.

The abstract language allows shorter, more intuitive formalisations of properties of the system. Often, agent notions or properties pertain to a period in time, which makes the detailed formulae describing them long and rather cumbersome. In AL, these temporal notions can be captured in a single abstract predicate, yielding a compact and clear representation. To illustrate this, we look at the following abstract formula:

$$\text{ActionEff}(A, l, t, \mathcal{M})$$

and its interpretation in the detailed language:

$$\begin{aligned} & \exists t_1 < t \exists t_2 \geq t : \\ & \oplus \text{state}(\mathcal{M}, t, \text{output}(\text{EW})) \models l \wedge \\ & \oplus \text{state}(\mathcal{M}, t_1, \text{input}(\text{EW})) \models \text{to\_be\_performed}(A) \wedge \\ & \otimes_{t_1, t_2} \oplus \text{state}(\mathcal{M}, t_2, \text{output}(\text{EW})) \models \text{ended}(A) \wedge \\ & \text{sit\_spec\_effect}(A, \text{state}(\mathcal{M}, t_1, \text{output}(\text{EW})|_{\text{world info}}, l) \end{aligned}$$

Proof structures sometimes can be made visible within the abstract language, as is the case for the proof of `COORD1`. As we showed in Section 3.4.3, the proof of `COORD1` from other coordination properties is done entirely within the abstract language, using only simple modus ponens and quantifier manipulation. By purely abstract proofs like this, complexity of the verification process is reduced considerably. More detailed parts of the verification process can be hidden in the detailed language, and show up in the abstract language only in the form of, abstractly formulated, reusable properties. Examples of such properties are `COORD0` and `COORD5`. The proofs of these properties, which are formulated in the abstract language AL, mainly use other properties of the agent and the environment phrased in AL. But in the proofs, the abstract notions are expanded into detailed ones (in TL), and these details are used for establishing validity of `COORD0` and `COORD5`. These proofs are long, involved and not extremely intuitive, but they only need to be done once. As we have proven `COORD0` and `COORD5` in a generic manner, these proofs remain valid for all reactive single agent systems built according to the generic architecture defined in Section 3.2. Therefore, when for a specific instantiation of the generic architecture action successfulness needs to be shown, there is no need to go into details anymore; the verifier can simply reuse the proofs we constructed and only focus at the abstractly formulated properties to get an idea of the demands on the agent and the world. Of course, the agent and world properties also have to be proved for the overall proof to be valid.

The languages we use in this chapter are similar to the languages used in situation calculus [90, 106]. A difference is that our languages allow explicit

references to temporal traces and time points. In [105], Reiter addresses proving properties in situation calculus. A difference with our approach is that we incorporate arbitrary durations in the decision process of the agent, and in the interaction with the world. Another difference is that in our case the induction principle works for a larger class of temporal models than the class of discrete models in his case.

Two roles are distinguished within the verification engineering process: the verification support developer, who *defines* libraries of reusable properties in the abstract language, and their related properties in the detailed language, and the verification engineer, who *uses* these libraries to actually perform verification for a system being developed in practice. This means that the verification support developer needs to possess a high level of formal skills, whereas the requirements for the verification engineer are less demanding.

In this chapter, we have been the verification support developer, as we created reusable properties and proofs for establishing action successfulness. Verification engineers can subsequently use these to verify specific reactive single agent systems interacting with a dynamic environment. Under realistic assumptions, such as action generation and action execution with duration, it is a complex problem to guarantee the successfulness of action executions. We addressed and clarified this problem using the approach described above. We defined a system (with two sub-systems) of reusable coordination properties. As we phrase the properties in the abstract language, the properties are much more accessible and explainable than their detailed counterparts are. The coordination properties found have become the beginning of a library of reusable properties that is being developed.

# CHAPTER 4

---

## An Abstract Model for Agents Interacting in Real-Time

---

*Time is a train  
Makes the future the past  
Leaves you standing in the station  
Your face pressed up against the glass*

*U2*

### 4.1 Introduction

If we look at traditional computer programs and agents, traditional programs are like vending machines (providing drinks and candy-bars), while agents are more like waiters. When you want something from a vending machine, you have to provide input, that is, to put money into the slot and push the keys that indicate the particular candy-bar you want to buy, and then the machine moves the candy-bar to its output slot. So, this machine simply computes a function from input to output, like traditional computer programs. There is no

interaction with other vending machines (“Hello machine–235, I’m presently out of cola; do you have some bottles for me?”), nor with the physical world outside the vending machine (vending machines don’t move, nor do they observe whether the person buying something looks happy or sad). Waiters are very different; their job is to interact with their customers. Thus, they will assist you when you have questions about the dishes on the menu, and they will adapt their actions to your preferences (if they’re good waiters). Waiters also communicate with colleagues (other waiters, the cook) and observe and act in the physical environment of the restaurant (they usually don’t trip over a handbag, when bringing plates to some table). Also, waiters can cooperate. If someone orders a very big birthday cake, two waiters could bring the cake to the table. The ability to interact with their environment makes waiters much more flexible, and thus more intelligent, than vending machines. On the other hand, it makes them more vulnerable as well; the actions of two waiters can interfere (they may bump into each other), or their actions can fail because of events in the restaurant (if a chandelier suddenly falls from the ceiling, the action of a waiter bringing someone a drink can fail). Like the waiters, agents are involved in an ongoing interaction with their environment, which consists of the physical world and other agents. On the one hand, this built-in ability to interact makes agents more powerful than other software entities, and on the other hand it makes them more vulnerable to disturbances of their activities.

One definition of an agent as given in [125] states:

“An agent is a computational entity such as a software program or a robot that can be viewed as perceiving and acting upon its environment and that is autonomous in that its behaviour at least partially depends on its own experience.”

So, an agent is situated in an environment which can consist of other agents and a physical or virtual world. For example, in robot soccer the environment of one agent consists of the other agents and the physical reality of the playing field and the ball. And a personal agent searching the internet for its owner is situated in the environment of the web.

In Chapter 2, we argued with many other authors that the benefit of the agent paradigm is that it provides new (anthropomorphic) concepts of a high abstraction level. The expressive power and intuitiveness of these concepts facilitate the design and construction of a new kind of computer systems. A multi-agent system consists of multiple agents, which can *interact* with the world and each other and which have a private internal state. The agents may be heterogeneous, and the system may be open to agents leaving or entering the system. Although each agent may have its own motivations, the agents

generally don't blindly act on these only. Agents also take into account the dynamics of the physical environment, where things may change all the time through actions of other agents or through natural events. Interaction can bring problems or provide opportunities and solutions. An action of an agent can fail because of an event taking place in the physical world, or because an action of another agent interferes with it; a condition that one agent is not capable of establishing is realised by cooperating with another agent, or it is established by an event.

*Agent programming languages* offer programming constructs that operationalise the mental concepts from the agent paradigm. Typically, agent programming languages provide statements for mental updates (of the beliefs, goals and intentions) [60, 101, 113]. Although agents offer the right level of abstraction for building the complex systems needed nowadays, and interaction is very much at the heart of the agent concept, interaction issues are not properly treated in many agent programming languages. More specifically, agent programming languages generally discard real-time issues such as interference of actions with duration and discrepancies between the actual state of world and the world state as the agent observes it. Typically, actions are atomic and take zero time, events from the environment aren't explicitly modelled and synergy of actions into group actions is impossible.

In this chapter, we will identify these and other aspects of agent interaction and incorporate them in an operational model of agents. The model we propose is a *skeleton programming language* with a *formal operational semantics*. When *using* the model, several parameters of the skeleton language have to be *instantiated*, in order to obtain a concrete agent programming language. Most of the flexibility is related to the internal processing of individual agents. We *abstract* from internal agent matters, in order to focus on agent *interaction* and on *real-time* aspects. This results in details of the inner agent being flexible, and shapable to the personal preferences of the system builder.

Many agent programming languages, such as AGENT-0 [113], AgentSpeak [101] and 3APL [60], abstract from an external real world. Agents in these languages maintain an internal database, that represents the state of the world. When the agent acts, this database is changed. The physical execution of an action is considered to be a *side effect* of the change in the belief base, and the underlying implementation has to take care of this execution. As the database always reflects the current state of the world, observations are implicit. Because there is no explicit world, the agent models of these languages also abstract from unexpected events taking place. Also, when multiple agents act, their actions can never interfere with each other, in case they only have private databases modelling mental states.

Another issue is the nature of actions. In our model, we model action duration by semantically equating each action with a sequence of atomic sub-actions. To determine the behaviour of a multi-agent system, we combine the effects of all sub-actions and events taking place at some time. In many other agent models, an action is an atomic state transformer, and its semantics is a priori given and independent of other actions or events taking place concurrently.

We think these choices abstract away many problems that can occur when building an actual system with multiple agents. Therefore, we have developed a programming language that allows for the full complexity of agents interacting with each other and the environment. In other words, the language doesn't abstract from real problems of complex open multi-agent systems in a dynamic environment. We have actions which take time, events that can unexpectedly change the state of the world, agents operating at different paces, mental states that don't have to reflect the present state of the world, and so on. Our target application is a system with a number of robots, situated in some environment. In such an application the dynamic nature of the world and interferences of actions are real problems. The previous chapter also provides evidence for this. There, we worked on verifying that a single agent situated in a dynamic environment successfully executes its actions, which have a duration. We had to exclude the occurrence of interacting events in order to be able to prove this. All situated agents have to deal with interaction issues, but our model is the only agent programming approach we know of that truly incorporates them. So, the contribution we make is *realism* with respect to the dynamics of interaction.

Although there is a proliferation of agent models, laid down in agent logics, architectures and programming languages, most agents perform the *sense–reason–act cycle*. First the agent *senses* its environment through observation and communication, then it *reasons* to update its local state with the new information and to decide which action(s) to perform next, and finally it *acts*, changing the world state, after which the cycle starts again. This could go on forever, or it could stop as soon as the ultimate objective of the agent is reached. In [92], it is shown how several architectures for deliberative agents employ some form of this cycle.

Programs in the agent programming languages we mentioned above also are executed in sense–reason–act cycles. The agent program only specifies the reasoning of the agent. The sensing, acting and cycling is built into the (informal or formal) semantics. A disadvantage of a semantic reason–act cycle is that it is not very flexible. To modify certain aspects of the cycle means to modify the semantics of the language, which is by no means easy. So, in our

programming language we have a *syntactic* cycle, which is programmed by the system builder and which specifies reasoning as well as sensing and acting. This gives freedom to adjust the processing in different rounds of the cycle to circumstances in the environment and mental state of the agent.

In order to abstract from details of the inner agent and provide maximal flexibility, our model has:

- \* A definable range of agent concepts. So, there is no fixed set of modalities (such as beliefs, desires and intentions) that is used to program agents. The set is chosen by the system developer.
- \* A flexible execution cycle. Our model only fixes that during every round of the cycle reasoning should precede interaction with the environment. As observation in our view is just a special kind of action, we call the cycle the *reason–interact cycle*. The agent programmer has the freedom to decide which part (if any) of the observed and communicated information the agent incorporates in its mental state, and also how the mental state of the agent determines the actions that are to be performed next.

By offering this flexibility, we allow the user of the model the freedom to fill in the internal agent according to his/her favourite agent theory or architecture.

In order to make the model more realistic with respect to interaction, we include:

- \* A *world*, which is common to the agents, but only partially controlled by them. So, unexpected changes to the world state, called *events*<sup>1</sup>, are allowed.
- \* Explicit *actions of observation*. So, the agents' view of the world as represented in its local state doesn't automatically match the state of the world.
- \* *Actions with duration*. We allow the result of an action to depend on whether other actions or events take place during the execution of the action.
- \* A treatment of the parallelism relevant to multi-agent executions, in which *real-time* processing is vital. In our semantic model, concurrent actions are treated as happening during intervals of time, which can overlap.

---

<sup>1</sup>We use the term 'event' in a different manner than other authors (e.g., [7]), where an event can be an agent-initiated action as well as a 'natural' change to the world, not caused by any agent in the system. In our view, events are changes to the world state, not caused by any agent.

Parts of actions thus take place simultaneously. This contrasts with the common interleaving semantics, where parallel actions are sequenced in a non-deterministic order. (See [4] for more explanation.)

- \* Both group actions and individual actions. Group actions are executed synchronously by the group members. We also define a relation between group actions and the individual actions of the participants. A group action is the result of the synchronised execution of individual actions by the group members.

The rest of this chapter is structured as follows. Section 4.2 informally explores several fundamental choices of the operational model. In Section 4.3 and 4.4, we give the syntax and semantics of the skeleton programming language, respectively. In Section 4.5 we prove that the syntax of the language guarantees appropriate semantical reason–interact cycles. Section 4.6 is an illustration of the new features of the agent model. In Section 4.7 we conclude.

## 4.2 Informal explorations

### 4.2.1 Cycles of reasoning and interaction

A multi-agent system in our agent model consists of a number of *agents* and a *physical world*, which is common to the agents and which they sense and act upon. Our model is most suitable for *embodied agents* or robots, as many complications of interaction are particularly manifest when there is a physical environment in which the agents operate. The agents can execute different sorts of actions. First, there are *physical actions*, which are actions that change the state of the physical world. Physical actions can be *single-agent* actions, or *group* actions. Obviously, single agent actions are performed by one agent, while group actions are done by a group of agents. Group actions, like lifting a table, are synchronously performed by the group members. In Section 4.2.3, we provide more intuitions about the relation between group actions and individual actions. Secondly, there are *observative actions* (performing an observation) and *communicative actions* (sending a message to another agent). Communication in the model is asynchronous. The third kind of actions are *reasoning actions*, which are actions for investigating and updating the local mental state of an agent.

One of the central ideas of our model is the *separation of internal and external agent aspects*. Internal aspects of the agent are the agent’s mental state, and the reasoning the agent performs. In this chapter (as in many chapters

of this thesis) we abstract from details of the inner agent. This is why several parameters of the flexible agent programming language we will define pertain to internal, mental, local agent features. External aspects of the agent are the agent's *interaction* with its environment, which consists of the physical world and the fellow agents. Observation, communication and physical action thus are external actions. And from the point of view of a certain agent, the state of the physical world and the mental state of other agents are external aspects. In our model, we take care that external agent actions only directly influence the external state, and internal actions only modify the internal state of that agent. So, when an agent performs a physical action, its model of the world which it stores in informational attitudes in its mental state isn't immediately changed. For example, if an agent throws a brick towards a window, it won't believe that the glass is broken immediately after it finishes the action. This can only happen after the agent has performed some observations regarding the effects of its action. Also, a reasoning action never directly modifies the state of the physical world. For example, if an agent adopts the intention to do an action, this doesn't mean that the action will start automatically. The agent must explicitly decide to execute the action before it will be performed.

In other agent programming languages, this is different. For example in 3APL [60], all the agent does is updating its belief and goal bases. Here, goals are the atomic or composite actions the agent will perform shortly. Adding a basic action as a new goal automatically can lead to execution of the action. And when the action is executed, the belief base is immediately updated with beliefs regarding the effects of the action. The reason for these choices is that the authors of 3APL want to provide a programming language of a high abstraction level, with the main focus on the internal reasoning of individual agents, and abstracting from details of interaction with the environment.

In contrast to 3APL, our operational model focuses on environment interaction. We opted for the separation between the inside and outside of an agent because this allows us to zoom in on details of interaction without clutter caused by internal reasoning processes of agents.

Each agent in the model has a local state, which consists of two parts: the *sense buffer* and the *mental state*. The *mental state* belongs to the internal part of the agent. It contains a set of formulas, describing the agent's present motivations and information. We don't fix a set of mental operators for the agents, such as the classical B(elief) D(esire) I(ntention) combination. The system builder can choose the set of modalities the agents can use, as this set is one of the parameters of the skeleton programming language. The *sense buffer* is a part of the state of the agent where results of observation and information communicated by other agents is received. Thus, observation and communica-

tion don't directly cause a change to the private mental state of the agent. The agent can investigate its sense buffer, and then decide whether newly arrived observed or communicated information should lead to a change to its mental state (for example its beliefs). This way, the agent can for example discard messages sent by agents it judges to be unreliable. This has similarities with the way Zhisheng Huang treats incoming information in his thesis [65]. The sense buffer is the frontier between the internal and external agent aspects; the agent receives external information in its sense buffer, and subsequently tests the sense buffer, using internal reasoning actions.

As stated in the Introduction, a natural way to envisage the processing of agents is that agents execute sense–reason–act cycles. Agents obtain new information by sensing (observing, receiving messages from other agents), then they update their mental state and on the basis of this updated state decide what to do next, after which they perform the actions. In the model we propose here, agents also execute cycles, though there is a small difference between the cycles in our model and the common conception of sense–reason–act cycles.

We choose to classify the performance of observations and the sending of messages as external actions, together with physical actions. We call actions of communication and observation and physical actions *Interactions*, and internal, reasoning actions *Intractions*. The cycle agents execute in our model is called the *reason–interact cycle*, and each round of this cycle consists of a first phase in which the agent executes intractions on its local state, and a second phase in which the agent performs interactions with its environment. We term the first phase of the reason–interact cycle the *reasoning phase*, and the second phase the *interaction phase*, for obvious reasons. During reasoning, the agent can modify its mental state, based on newly received observation results and communicated messages from the sense buffer; for example, it can revise its beliefs or goals or adopt new intentions. Also, the agent decides which actions are to be done next. These actions can be physical actions, that will change the state of the world, or they can be actions of observing or communicating with other agents. Then, the interacting phase of the cycle begins, in which the interactions are executed.

When we compare our skeleton programming language to other languages, our execution cycle provides more freedom. For example, in other languages the receipt of messages and observation results and the processing of these into new beliefs happens outside the range of the agent program, in the control cycle of the interpreter. So, there is no way to change the agent program in order to achieve a different policy with regard to incoming communication. If you want to change this, you have to change the semantics of the programming language and the interpreter of the language. In our language, almost everything

is programmable. Most programs will have a part that fixes how incoming information is processed into beliefs or other mental formulas, as well as parts for other agent features. So, when you want to change some characteristic of the agent, you can simply change its program.

This is an advantage of our model, as it makes the agents very flexible. But this choice leads to an execution cycle which is defined quite differently from the cycles in other approaches. A typical sense–reason–act cycle in other languages looks like this:

- \* Receive and process incoming messages; update the beliefs.
- \* Revise the goals and/or commitments, according to the agent program.
- \* Execute all commitments which are valid now.

The agent program only specifies the second phase of this cycle. When the cycle is in this second phase, the program of the agent is interpreted, taking into account the mental state. Then, the execution phase takes place, in which the effects of the action(s) are computed and added to the mental state. While the results of actions that change the world state immediately update the belief base of the agent, the agent mostly doesn't perform explicit observations, and the sense–part of the sense–reason–act cycle isn't needed. The receipt and processing of new information and the execution of the commitments are taken care of by the semantics of the agent program as laid down in the interpreter. This allows the program to be compact. In our cycle, the program influences all three phases, and thus it is more elaborate.

Details of the cycle differ across the different programming languages. For example, sometimes the agent executes all possible mental updates, after which it performs all actions selected for this time point, as in AGENT-0 [113], and sometimes the agent takes one reasoning step only, after which it executes one basic action, as in 3APL [60]. These choices are fixed in the semantics of the programming languages. This makes these programming languages less flexible than the programming language of our model. The cycle in our programming language is defined syntactically, by the way the program is constructed; programs always are an iterated alternation of phases of internal reasoning and interaction with the external environment.

In our model, physical actions have duration. Each action takes a fixed number of ticks of a global clock (which is positioned in the world, which is common to all agents). The presence of a global clock might seem a weakness of the model, as it forces the agents to be perfectly synchronised. In Section 4.2.3 we will explain the remedy against this unwanted side-effect. We assume

that reasoning, observing and communication take little time in comparison to physical action. So, in our model reasoning actions and communicative and observative actions each take one time unit. As the system builder has to choose the precise duration of a time unit, he or she has to take care that each communicative, observative and reasoning action will be completed within one time unit.

### 4.2.2 Multiple parallel behaviours?

In our model, the two phases of the reason–interact cycle take time. In the reasoning phase, the agent uses this time only to reason, and in the interaction phase, it only performs external actions. While executing the reason–interact cycle, the agent can perform a number of interactions with its environment concurrently or be engaged in different parallel threads of reasoning. But as internal and external operations are strictly separated, it is not possible to reason and interact in parallel. It might seem a waste of resources not to reason at all during the interaction phase, and not to act at all during the reasoning phase. This is why in an earlier version of our agent model, one agent could execute multiple, parallel reason–interact cycles. We call one cycle a *behaviour*, and allowed the agent to have several behaviours. The idea was that an agent can reason about what to do next in one behaviour while it is busy doing actions from another behaviour. Also, interactions from different behaviours could be executed in parallel.

But multiple behaviours are only possible when the agent/robot has several entirely disjoint interests. The actions in two parallel behaviours should not interact, and the (observed, communicated) information used in two behaviours should be different. Also, no element of the mental state should be accessed by two different behaviours. This is because two behaviours are not synchronised in any way, so only when they are completely independent, they can be concurrently executed in a safe way. Whenever there is potential interaction in two behaviours, they should be integrated into one behaviour, such that more control is possible regarding the flow of mental updates and interactions.

So, in order to justify multiple behaviours, we have to think of some example of an agent where the agent has at least two completely independent interests. This is not very easy. The physical actions of the two behaviours should use different actuators and alter different parts of the physical reality. An example would be an agent who is moving from one place to another while at the same time juggling three balls. This kind of behaviour is nice for amusement, but is not extremely useful. When thinking of the soccer robots, no parallel independent behaviours in one robot seems possible. Everything these robots

do and reason about is connected with the position of the ball relative to the agent. For example, if a soccer robot is moving forward towards the ball and simultaneously turning its camera to get new information on the position of the adversaries, then both these actions are ultimately aimed at getting the ball into the goal of the opponent. Information from the camera could influence the moving of the robot. Thus, concurrent execution of these actions makes sense if this happens in a controlled manner, as is only possible if both actions are part of the same behaviour.

In fact, it is even difficult to think of parallel behaviour in humans. I am able to think about what to have for dinner while I cycle home, but this is not a good example as cycling is done without thinking at all and the actions following my reasoning about dinner (cycling to the shop) might interact with the behaviour of cycling home. I can type and listen to music simultaneously, but this is no good example as listening to music doesn't incorporate any (physical) action. I can smoke a cigarette while walking to the supermarket, but this is only possible because I don't have to think about how to smoke a cigarette. Usually, when a human is capable of two parallel behaviours, one of them is purely based on reflexes, needing no conscious mental processing at all. This view is supported in the cognitive science textbook [56], by Hendriks, Taatgen and Andringa. On pages 114–115 the authors describe *controlled* processes, which are cognitive processes requiring conscious attention and taking up a lot of the total capacity of information processing. They state that it is hard to simultaneously perform multiple controlled processes. Doing two things in parallel is well possible when at most one of the processes is a controlled process, and the other(s) are *automatic* processes, taking up little information processing capacity. This is why it is hard to do something else when you're learning a new skill (pages 122-123), such as dancing. Only when you can dance 'automatically', it becomes possible to have a conversation with your dancing partner. We conclude that cognitive science doesn't inspire us to give agents multiple behaviours, unless only one of these behaviours contains internal reasoning actions. As in general behaviours contain both reasoning and acting, we decide to give each agent only one behaviour.

As another argument, imagine a robot performing two totally independent activities, like doing your dishes and sorting your mail. This robot is likely to be conceived as containing *two* agents, one dedicated to the dishes and the other to the mail, sharing a single body (meant as vehicle of sensors and actuators; most likely, sensors and actuators are not shared between the two agents).

Summarising, we conclude that we don't have convincing arguments for multiple behaviours in one agent. Agents in our model thus execute a single behaviour, which also has the advantage that the semantics of the programming

language will be less intricate.

### 4.2.3 Group action and individual action

Agents can perform individual physical actions and group physical actions. The sets  $\mathcal{A}_s$  of individual actions and  $\mathcal{A}_g$  of group actions are parameters of our skeleton programming language, as the actions of the agents are domain-dependent. One of the objectives of the agent model is that actions have duration, and that two (or more) actions that are executed during overlapping time frames could interfere, that is, yield effects different from the effects that would result if the second action starts after the first is finished. To reach this objective, we have to find a suitable real-time semantic model. An interleaving semantics, which interprets parallel actions as taking place one after the other, is not right for this purpose. In this subsection, we will sketch some features of the semantic model we use for the skeleton programming language.

Physical actions take time. For individual actions, we model this by relating each action to a fixed sequence of atomic sub-actions, which each takes one time unit.<sup>2</sup> The sub-actions never occur in a program; they only play a role in the semantics. The precise length of the time unit is chosen by the system builder. The set  $\mathcal{A}_{sub}$  is the set of all atomic sub-actions. For example, if an action  $c$  takes three units of time, then it is semantically related to the sequence  $c_1; c_2; c_3$ , where  $c_1, c_2, c_3 \in \mathcal{A}_{sub}$  are the sub-actions of  $c$ . When constructing the execution traces of an agent program containing the action  $c$ , three consecutive transition steps are generated for the three sub-actions, if the action is executed successfully. As each sub-action can either succeed or fail, depending on circumstances in the environment, we attach a sign to each sub-action to indicate this. Thus, there are four possible executions of action  $c$ :  $c_{1+}; c_{2+}; c_{3+}$ ,  $c_{1+}; c_{2+}; c_{3-}$ ,  $c_{1+}; c_{2-}$  and  $c_{1-}$ . When a sub-action fails, the execution of the action is over; the remaining sub-actions are discarded of.

In case two agents are executing actions, the atomic sub-actions of these actions aren't interleaved, as is usual in the semantics of many programming languages. Instead, the sequences are indivisible but can overlap in time, as Figure 4.1 illustrates. In this figure, we show part of the trace of an agent system, where two actions,  $a$  and  $b$ , successfully take place. The first action takes five time units and the second takes four time units, as modelled by the number of sub-actions of  $a$  and  $b$ . An interleaving semantics could result in execution of a sequence like  $a_1+b_1+a_2+a_3+b_2+a_4+b_3+b_4+a_5+$ , where the atomic

---

<sup>2</sup>This has similarities with the way Cohen and Levesque interpret actions in their landmark paper [21].

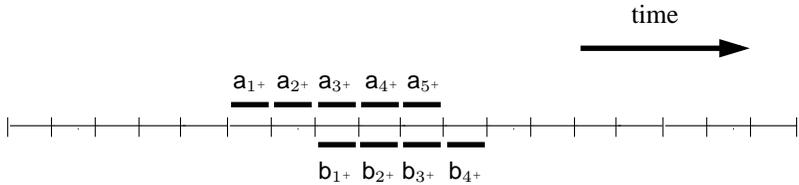


Figure 4.1: Overlapping action executions

steps of the actions are shuffled in a linear order. Steps are thus not performed simultaneously, in the same time unit. The semantics we will use for the operational model is different. In this semantics, which is depicted in Figure 4.1, the sequences of sub-actions overlap in time; interference of the effects of these actions is possible. For each time unit, the state change these actions cause is computed by taking the combined effects of the sub-actions taking place at that time. This semantics is called a *step semantics*; we used ideas from [91].

An alternative way to formalise actions with duration is the use of time intervals [2] associated with actions, as is done in [110]. Then, the actions from Figure 4.1 would look like  $a[10, 15]$  and  $b[12, 16]$ . Though this may be more elegant, this formalisation offers no straightforward manner to arrive at the state transformation caused by interfering actions. Using step semantics, a state transformation can be computed for each time unit as will be shown in Section 4.4.3. This explains our choice for sequences of sub-actions to model action duration.

In our semantical model as presented above, a global clock is present, which ticks when time units elapse. The presence of a global clock is not a realistic assumption for multi-agent systems, as the agents may be very heterogeneous and have local clocks ticking at different paces. We don't know anything about the local clocks of the agents and the relation of the different local clocks to the global clock. And we don't even *want* to know anything about this, as the agents in a multi-agent system can be built by different people, using different formalisms and conventions, and we want to be able to cope with this. But we had to add the global clock in the world to be able to give a semantics to the interacting behaviours of multiple agents. To prevent this global clock from setting the pace of the entire system, turning it into a synchronised distributed system, we allow arbitrarily long delays preceding each interaction with the environment. In this way, it is possible that an agent doing a physical action based on some observations it did earlier may find the world totally different from its observations, because one or more faster agents

have done some actions in the time it was still busy reasoning and starting the execution of the action.

For group actions, there are additional matters to consider. A group action actually is a multi-set of synchronised individual actions. For example, when four agents are doing the group action `LiftTable`, then each of these agents is lifting one table-leg. They could do this perfectly synchronised, but the action will probably still succeed when one agent starts lifting one time unit before the others. Also, the agents can lift the table together by lifting a side each, instead of a leg.

In our model, each group action has a fixed number of participants. Each possible realisation of a group action is laid down in a *group action scheme*. If  $b \in \mathcal{A}_g$  is a group action with  $k$  participants, then a group action scheme  $A \sqsubseteq \mathcal{A}_{sub}^*$  for  $b$  is a multi-set with  $k$  elements. A group action scheme contains a sequence of atomic actions for each of the participants. To represent different starting times of the actions of the participants, some of the sequences have leading or trailing skip statements. A skip-statement means that the agent does nothing for one unit of time. We suppose  $skip \in \mathcal{A}_{sub}$ , to technically allow skips in group action schemes. The duration of a group action can vary across the different group action schemes realising it. Therefore, we refer to the duration of a particular group action scheme, and not to the duration of the group action itself.

To illuminate this, we extend the example of `LiftTable`  $\in \mathcal{A}_g$ . Lifting a table is done by four agents, which each lift a table-leg. `LiftLeg`  $\in \mathcal{A}_s$  is the individual action the four agents perform. This action takes three units of time; during each unit of time, the table-leg can be lifted 15 cm. So, `LiftLeg` is related to the sequence `LiftLeg0-15`; `LiftLeg15-30`; `LiftLeg30-45`. Here is a group action scheme for `LiftTable`:

```
[LiftLeg0-15; LiftLeg15-30; LiftLeg30-45; skip; skip,
 skip; skip; LiftLeg0-15; LiftLeg15-30; LiftLeg30-45,
 skip; LiftLeg0-15; LiftLeg15-30; LiftLeg30-45; skip,
 skip; skip; LiftLeg0-15; LiftLeg15-30; LiftLeg30-45]
```

This group action scheme has a duration of five time units. In other group action schemes for `LiftTable`, the `LiftLeg`-actions could have different relative delays, or be perfectly simultaneous. In this last case, the group action scheme would take only three time units. Also, there can be group action schemes where the four agents each lift a side instead of a leg.

Again, group action schemes are not directly used in the syntax of agent programs. But when semantics is given to an agent program, and a group action

is encountered, then this group action is replaced by a non-deterministically chosen individual contribution from one of the applicable group action schemes. Like ordinary individual actions, this contribution can fail at any point during execution, so the sub-actions are annotated with signs for success or failure. A sub-action annotated with ‘-’ is executable in case the circumstances in the physical world prevent successful execution and execution will thus result in an undesirable world situation. For example, if the robot arm is not holding the table-leg, then the sub-action LiftLeg0-15 will fail, so LiftLeg0-15- is executed and the table-leg will stay where it was. If the world circumstances allow successful execution of a sub-action, then the variant annotated with ‘+’ is executable.

We will illustrate the use of group action schemes. Discarding success and failure for a moment, the group action (LiftTable,  $\{i_1, i_2, i_3, i_4\}$ ) in the program of agent  $i_2$  can be replaced (when giving semantics to the program) by

LiftLeg0-15; LiftLeg15-30; LiftLeg30-45; skip; skip or by  
 skip; LiftLeg0-15; LiftLeg15-30; LiftLeg30-45; skip,

or yet another element from the group action scheme. If there are more group action schemes for LiftTable, then action sequences from these are also options. If we add annotations to the sub-actions, then three possible extensions for  $i_2$  of LiftTable (based on the second sequence above) are:

- \* skip+; LiftLeg0-15-
- \* skip+; LiftLeg0-15+; LiftLeg15-30-
- \* skip+; LiftLeg0-15+; LiftLeg15-30+; LiftLeg30-45+; skip+

The skip action never fails, as there is virtually no physical circumstance imaginable in which doing nothing couldn’t be executed (except destruction of the agent or the end of the world; we ignore these somewhat unlikely options). So, we equate skip+ with skip. The first extension above fails after the first step of lifting the table-leg, the second after the second step and the third extension is a successful execution of the whole group action contribution.

A group action can only succeed if there is a group of agents, each executing an individual contribution to the group action, that fit together according to one of the schemes of the group action. If these conditions don’t hold, for example because there are not enough agents contributing to the group action, or because one of the contributions of the group members fails in the middle, then the individual attempted contributions could still succeed, but the synergetic effect will not occur. For example, when four agents attempt to lift a table

on which there is a bowl of soup, but the LiftLeg-actions of two adjacent agents fail in the middle, then the other agents can still lift their table-leg, but the soup will spill over the bowl as the table is not lifted evenly. In Subsections 4.4.1 and 4.4.3, we go into semantic details of these issues, and in Section 4.6 we give examples to illustrate these technical details.

In our view on group actions, we look at agent behaviour from an external point of view, matching our focus on agent interaction (and not on internal agent aspects). If there are several agents that perform individual actions matching a group action scheme, then we perceive the activity of these agents as a group action, even though they might be doing this combination of actions purely by coincidence, without any coordination. We are only interested in the synergy of particular combinations of individual actions, and not in the presence or absence of internal motivations that agents establish in order to coordinate their activity. In order to purposely obtain the conditions for group action success, meaning that there is a group of agents agreeing on the group action scheme to be used, the task delegation and the proper time to start the action, the group members need to coordinate their activities. Possibly, they might come to joint mental attitudes, which formalise the agreement of the group on details of the group action. Though this is very important, this chapter won't go into coordination; it is interesting for future research to link the model defined in this chapter and theories of agent coordination. In the next two chapters, we will address coordination aspects. We design a programming language and a coordination language with special features for coordinating agents in Chapter 5 and 6, respectively.

Note also that it doesn't matter which agent in the group performs some action sequence from a group action scheme. Our present formalism doesn't allow specifying that certain individual contributions to a group action can only be made by agents that have specified abilities and/or authorisations. In other words, we have no roles; all group members are equal. Future research could also address this. We refer to research by Virginia Dignum and others [30] for some work on roles and agents.

### 4.3 Syntax

We briefly summarise the sketch of agents given in the previous section, before we start describing the syntax of the skeleton programming language. Each agent has a *program* and a *state*. The state of the agent has two parts: the *sense buffer*, where observed and communicated information is received and stored, and the *mental state*, which contains a set of mental formulas. The program

contains two kinds of actions, namely internal reasoning actions, and external actions of interaction with the environment. The program explicitly specifies the reason–interact cycle, and thus it is an iterated statement. A *multi-agent system* consists of a set of agents and an external world, executing in parallel.

### 4.3.1 Basic sets

The basic building blocks of agent programs come from the following sets:

- \*  $\mathcal{P}$  = the set of propositional atoms formalising environment conditions
- \*  $\mathcal{I}$  = the set of agent identifiers
- \*  $\mathcal{M}_f$  = the set of agent modalities which apply to formulas ( $f$  refers to formula)
- \*  $\mathcal{M}_a$  = the set of agent modalities which apply to actions ( $a$  refers to action)
- \*  $\mathcal{A}_s$  = the set of actions agents perform individually ( $s$  refers to single agent); we assume  $\text{skip} \in \mathcal{A}_s$
- \*  $\mathcal{A}_g$  = the set of actions agents perform groupwise ( $g$  refers to group of agents)

The set  $\mathcal{P}$  forms the basis of the logical language we will use to describe the world situation and to phrase agent properties. Examples of atoms in  $\mathcal{P}$  are *large\_obstacle\_ahead* and *agent\_James\_unreliable*. As the last atom suggests, a first order language is more suitable for phrasing properties of the agent system. We opt for a propositional language only for reasons of simplicity; the skeleton programming language can easily be extended into the first-order direction.

The set  $\mathcal{I}$  doesn't need any explanation.

$\mathcal{M}_f$  and  $\mathcal{M}_a$  allow the programmer to choose which mental attitudes the agent employs. The mental state of an agent consists of a number of formulas, which (for example) represent the beliefs, goals and desires of the agent held at that moment. Each category of formulas has its own modal operator. The set of modalities is flexible. So, the programmer can decide whether or not the agents should have desires, goals or intentions. The programmer is also allowed to introduce completely new motivational and informational attitudes.

To instantiate the skeleton language, the programmer specifies two *modality sets*,  $\mathcal{M}_a$  and  $\mathcal{M}_f$ .  $\mathcal{M}_a$  contains the modal operators which apply to actions

(for example intention or request), and  $\mathcal{M}_f$  contains the operators which apply to formulas of logic (for example belief or desire). Formulas constructed with these modalities are called *mental formulas*; they can be part of the mental state. For example, by including the concepts belief, desire, goal, and intention in the modality sets, BDI-agents can be programmed. As another example, reactive agents are obtained if the modality sets are both empty, because then the agents can only decide what to do next on the basis of information they received by observation and communication. The modality sets can also contain less conventional operators. For example, if one agent has authority over other agents in the system, a modal operator for commanding to do some action could be an element of  $\mathcal{M}_a$ , such that the boss agent can send messages to its subordinates ordering them to do something. As another example,  $\mathcal{M}_f$  could contain an operator for expectation, such that agents can store and reason about their expectations, for example regarding the effects of actions they perform.

The programming language contains two *mental action constructors*, *ins* and *del*, for inserting into and deleting from the mental state. For example, if  $B \in \mathcal{M}_f$ , representing belief, then  $B_i$  is the belief operator for agent  $i$  and  $B_i(\varphi)$  (meaning “agent  $i$  believes  $\varphi$ ”) is a formula that can be part of the mental state of agent  $i$ . Examples of mental actions are *ins*( $B_i(\varphi)$ ) and *del*( $B_i(\varphi)$ ).

Mental modalities basically are just labels. To give them their intended semantics, mental formulas have to validate a set of axioms. The system developer is responsible for this.

There is another important basic set, namely  $\mathcal{E}$ , the set of events. These are changes in the world state not caused by any of the agents in the system. We didn’t mention  $\mathcal{E}$  in our list of basic sets, because events have no role in the syntax of agent programs, as they are not under control of the agents. The agents can observe the effects of events, and these observations can influence the processing of the agents.

### 4.3.2 Auxiliary languages

In order to define the syntax of the programming language, we need several auxiliary languages:

- \*  $\mathcal{L}_w$ , the set of formulas describing the state of the world, based on  $\mathcal{P}$ .
- \*  $\mathcal{L}_{sb(i)}$ , the language of the *sense buffer* of agent  $i$ . This language defines the set of formulas which can be present in the sense buffer. Basis of the language are formulas like  $Od_i(\varphi)$  and  $Cd_i(\varphi)$ , meaning that  $\varphi$  is observed by  $i$  or communicated to  $i$ , respectively.

- \*  $\mathcal{L}_{m(i)}$ , the mental language, that is, the set of formulas that are allowed in the mental state of agent  $i$ . These formulas are constructed with the mental operators from the flexible modality sets  $\mathcal{M}_f$  and  $\mathcal{M}_a$ .
- \*  $\mathcal{L}_{rep(i)}$ , the language of (internal) representations of agent  $i$ . Here “representations” is meant to cover both sensory and mental information. A formula of this language can describe a property of the combination of the mental state and the sense buffer of the agent.
- \*  $\mathcal{L}_s$ , the language describing properties of the whole multi-agent system, that is, of the environment and of internal representations of the agents. Formulas from this language can be the subject of the reasoning of the agents.
- \*  $\mathcal{A}_o$ , the set of all actions of observation.
- \*  $\mathcal{A}_{c(i)}$ , the set of all actions of communication that agent  $i$  can perform.
- \*  $\mathcal{A}_{m(i)}$ , the set of mental update actions of agent  $i$ .
- \*  $Interactions_i$ , the set of composite actions of agent  $i$ , constructed only from actions of observation and communication and physical actions.
- \*  $Intractions_i$ , the set of composite actions of agent  $i$ , constructed only from mental actions.

The auxiliary languages are defined by mutual recursion. As the reader may have noted, some of the languages are parameterised with an agent identity (for example  $\mathcal{A}_{c(i)}$ ), while other aren’t (for example  $\mathcal{A}_o$ ). The reason for this distinction is that the parameterised auxiliary languages contain agent-specific phrases, while the other languages don’t. This will become clear when we give the formal definitions below. As explained in Section 4.2.1, we choose to have a strict separation between internal, mental operations (testing, adding and deleting information) and external, interacting operations (observing, communicating, acting). Physical actions only change the state of the world and don’t directly affect mental states of agents; for this, the agents must observe first. And even observations and communications don’t directly affect the mental state. They arrive in the sense buffer, and the agent can examine this and decide to change its mental state then. Mental actions on the other hand only change mental states. Agents test their sense buffers, and decide to update their mental state by inserting or deleting formulas; by evaluating their mental state, they can decide to perform interactions with the environment. So,

the agent programmer has to supply an attention mechanism, to decide when to observe certain world conditions, and also a relevance evaluation mechanism, which judges the relevance of observed and communicated information. Usually, agent programming languages contain a built-in mechanism that automatically updates the state of the agent when new observed or communicated information arrives. Our skeleton programming language offers the freedom to create any mechanism deemed suitable, at the cost of more programming effort.

We introduce the two sets of composite actions  $Interactions_i$  and  $Intractions_i$  to be able to form agent programs which work in reason–interact cycles, as explained in Section 4.2.1. For programs, this means that it is forbidden to compose physical actions (including observation and communication) and mental actions in an arbitrary manner. In order to accommodate this, these two kinds of actions are separated into  $Interactions_i$  and  $Intractions_i$ . The set  $Interactions_i$  contains all external actions, that is, actions of interaction with the environment, while  $Intractions_i$  contains all internal, mental actions.

The first auxiliary language we define is  $\mathcal{L}_w$ . Formulas from this language describe properties of the external world and of the agents, and form the subject of much of the reasoning of the agents.

#### DEFINITION 4.1 ( $\mathcal{L}_w$ )

The set  $\mathcal{L}_w$  of formulas describing states of the environment is the smallest set containing:

- \*  $p$ , where  $p \in \mathcal{P}$
- \*  $\varphi \wedge \psi$ ,  $\neg\varphi$ , where  $\varphi, \psi \in \mathcal{L}_w$

The other logical connectives are defined in terms of  $\wedge$  and  $\neg$ , as usual. Examples of formulas in this language are  $window\_dirty \wedge \neg other\_agents\_present$ ,  $\neg foggy \rightarrow window\_dirty$  and  $agent\_Al\_stupid \vee agent\_Al\_brought\_passport$ . So, formulas of  $\mathcal{L}_w$  can describe states of the external world, as well as properties of agents. Note that the expressiveness of  $\mathcal{L}_w$  depends on the set of propositional atoms  $\mathcal{P}$ . Formulas from  $\mathcal{L}_w$  can be observed and be the subject of mental modalities from the set  $\mathcal{M}_f$ . For example, if there is a belief modality  $B$  in  $\mathcal{M}_f$ , then  $B_i(\neg foggy \rightarrow window\_dirty)$  is a well-formed formula, meaning that agent  $i$  believes that, in case it is not foggy outside, the window is dirty.

DEFINITION 4.2 ( $\mathcal{L}_{sb(i)}$ )

The set  $\mathcal{L}_{sb(i)}$  of formulas which can be present in the sense buffer of agent  $i$  is the smallest set containing:

- \*  $Od_i(\varphi)$ , where  $\varphi \in \mathcal{L}_w$
- \*  $Cd_i(\varphi)$ , where  $\varphi \in \mathcal{L}_{rep(j)}$ ,  $j \in \mathcal{I}$ ,  $j \neq i$
- \*  $\varphi \wedge \psi$ ,  $\neg\varphi$ , where  $\varphi, \psi \in \mathcal{L}_{sb(i)}$

The formula  $Od_i(\varphi)$  means that agent  $i$  has observed that  $\varphi$  is true of the environment, and  $Cd_i(\varphi)$  means that it has been communicated to agent  $i$  that  $\varphi$  holds of the state of the sending agent. The identity of this agent is coded in the formula  $\varphi$ , as will become apparent when we define  $\mathcal{L}_{rep(i)}$ , and as the following example shows.  $Od_i(window\_dirty) \wedge Cd_i(B_j(\neg window\_dirty))$  is a formula of  $\mathcal{L}_{sb(i)}$ , meaning that agent  $i$  has seen that the window is dirty, while agent  $j$  has told him that he believes that the window isn't dirty. We subscript the formulas  $Od_i(\varphi)$  and  $Cd_i(\varphi)$  with the identity of the agent because the formulas can be communicated to other agents, and the subscript then clearly indicates the origin of the information. Observed or communicated information in the sense buffer of an agent doesn't have to be true; observations may be unreliable, and agents that sent messages could have lied. Also, information can get outdated, when the agent doesn't receive new observations and communications often enough.

The following definition refers to the language  $\mathcal{L}_s$  which we will formally define in Definition 4.5.

DEFINITION 4.3 ( $\mathcal{L}_{m(i)}$ )

The mental language of the agent  $i$ , that is, the logical language that agent  $i$  employs to express its internal mental state is the smallest set containing:

- \*  $C_i(\varphi)$ , where  $C \in \mathcal{M}_f$ ,  $\varphi \in \mathcal{L}_s$
- \*  $C_i(\alpha)$ , where  $C \in \mathcal{M}_a$ ,  $\alpha \in Interactions_i$
- \*  $\varphi \wedge \psi$ ,  $\neg\varphi$ , where  $\varphi, \psi \in \mathcal{L}_{m(i)}$

All mental formulas use operators that are associated with concepts from the flexible modality sets. So, if  $\mathcal{M}_f$  contains  $D$ , formalising desire, then agent  $i$  can have formulas of the form  $D_i(\varphi)$  in its mental state. Mental operators from  $\mathcal{M}_f$  can be applied to any formula describing the state of the overall agent system, as will become apparent when we define  $\mathcal{L}_s$ . And the arguments to operators from  $\mathcal{M}_a$  are composite external actions, as agents can for example intend to do a couple of actions one after the other. Concepts from  $\mathcal{M}_f$  and

$\mathcal{M}_a$  are subscripted with agent identities, in order to discern between mental formulas from different agents.

DEFINITION 4.4 ( $\mathcal{L}_{rep(i)}$ )

The local logical language of agent  $i$ , used to describe properties of the combination of the sense buffer and mental state, is the smallest set containing:

- \*  $\varphi$ , where  $\varphi \in \mathcal{L}_{sb(i)} \cup \mathcal{L}_{m(i)}$
- \*  $\varphi \wedge \psi, \neg\varphi$ , where  $\varphi, \psi \in \mathcal{L}_{rep(i)}$

Agent  $i$  can test and communicate any formula from  $\mathcal{L}_{rep(i)}$  in its program. An agent thus doesn't have to choose between focusing on information in the sense buffer or information in its mental state; it can test or communicate formulas describing a property of the union of these two parts of the agent's state. An example of such a formula is  $G_i(\neg window\_dirty) \wedge Od_i(window\_dirty)$ , where we assume  $G$  (denoting goal)  $\in \mathcal{M}_f$ . Assuming  $B$  (denoting belief) is also present in  $\mathcal{M}_f$ , another example of  $\mathcal{L}_{rep(i)}$  is  $B_i(G_j(at\_tropical\_island))$ , which in fact also is a formula of  $\mathcal{L}_{m(i)}$ .  $G_i(Od_j(flowers\_on\_table)) \wedge \neg Cd_i(Od_j(flowers\_on\_table))$  states that it is a goal of  $i$  that  $j$  has observed that there are flowers on the table, but  $j$  hasn't told  $i$  yet that it saw them. As a last example of the contents of  $\mathcal{L}_{rep(i)}$ ,  $Cd_i(Cd_j(\varphi))$  is a well-formed formula. This expresses that agent  $i$  heard from agent  $j$  that agent  $j$  heard something from some other agent, so this means agents can *gossip*. They can communicate about what was communicated to them. Also, they can talk about what they *saw*.

The basis of  $\mathcal{L}_{rep(i)}$  consists of formulas that are an application of some operator  $X_i$  to some other construct. So, for each well-formed formula, it is always clear of which of the agents the state is described by the formula. This feature is the reason why the identity of the sender of message  $\varphi$  is not indicated in  $Cd_i(\varphi)$ . The formula  $\varphi$  clearly indicates the sender of the message.

DEFINITION 4.5 ( $\mathcal{L}_s$ )

The set  $\mathcal{L}_s$  of formulas describing the state of the overall multi-agent system is the smallest set containing:

- \*  $\varphi$ , where  $\varphi \in \mathcal{L}_w \cup \bigcup_{j \in \mathcal{I}} \mathcal{L}_{rep(i)}$
- \*  $\varphi \wedge \psi, \neg\varphi$ , where  $\varphi, \psi \in \mathcal{L}_s$

Formulas from  $\mathcal{L}_s$  are not constrained to a segment of the multi-agent system. These formulas can describe properties concerning several local agent states,

as well as the environment. From the definition it is clear that  $\mathcal{L}_w \subseteq \mathcal{L}_s$  and  $\mathcal{L}_{rep(i)} \subseteq \mathcal{L}_s$ , for each agent  $i \in \mathcal{I}$ .

DEFINITION 4.6 ( $\mathcal{A}_o$ )

The set of all actions of observation an agent can perform is defined:

$$\mathcal{A}_o \equiv \{\text{obs}(\varphi) \mid \varphi \in \mathcal{L}_w\}$$

An agent performing  $\text{obs}(\varphi)$  employs its sensors to establish whether  $\varphi$  is true in the environment. The agent can observe the external world and/or the (other) agents. The outcome of an observative action is influenced by the *observability* of the information. If, for example,  $\varphi$  concerns an object which is outside the range of the agent's sensors, then the observation won't yield the desired information. Other features of the environment might not be observable at all. For example, the formula *agent\_James\_unreliable* probably is not observable, though it is environment information. An agent can come to the conclusion that it believes this formula by interpreting James' behaviour.

In case agent  $i$  performs  $\text{obs}(\varphi)$ ,  $\varphi$  is true, and this is observable to  $i$  in the present world state,  $\text{obs}(\varphi)$  results in the formula  $Od_i(\varphi)$  being added to  $i$ 's sense buffer somewhat later. In case  $\varphi$  doesn't hold or the information is not observable, the result is that  $\neg Od_i(\varphi)$  arrives in the sense buffer.

DEFINITION 4.7 ( $\mathcal{A}_{c(i)}$ )

The set of all actions of communication agent  $i$  can perform is defined:

$$\mathcal{A}_{c(i)} \equiv \{\text{comm}(\varphi, j) \mid j \in \mathcal{I}, j \neq i, \varphi \in \mathcal{L}_{rep(i)}\}$$

An agent performing  $\text{comm}(\varphi, j)$  sends a message to agent  $j$ , containing the information  $\varphi$ . The communicating agent can send every formula which could be in its local state, independent of the fact whether it is currently true or not. So, agents are allowed to *lie*. If  $i$  performs  $\text{comm}(\varphi, j)$ , then the formula  $Cd_j(\varphi)$  arrives in the sense buffer of agent  $j$  some time later.

The actions of observation and communication update the sense buffers of the agents. In this chapter, the meaning of these actions stays rather abstract, as we don't go into the process of *revision* of the sense buffer. But work has been done in this direction. In [84], Van Linder, Van der Hoek and Meyer introduce actions for updating the mental state of agents, which can contain both knowledge and belief. The authors follow the lines of the well-known AGM framework as proposed by [1] and provide three actions to change the agent's minds, which are *expansion* for adding formulas to the beliefs, *contraction* for removing formulas from the beliefs and *revision* for revising the beliefs with

new information, yielding a consistent new state. These three update actions are given a rigorous semantics, which satisfies the AGM postulates (which state demands on belief revision results).

Communication in our model is asynchronous. This means there is no statement for listening and waiting for incoming messages, like there is in for example CCP [111]. In CCP, there is a *tell* statement for sending a message (which in case of CCP always is a value), and an *ask* statement for receiving a message. If we would adjust these primitives for our purpose, this would result in statements like  $\text{tell}(\varphi, j)$  (“tell that  $\varphi$  holds to  $j$ ”) and  $\text{ask}(\psi, i)$  (“be ready to receive any answer from  $i$  that implies  $\psi$ ”). Van Eijk took a similar approach in his thesis [41]. In synchronous communication, the *tell* and *ask* statements are executed synchronously; this means an asking agent has to wait till the appropriate agent will tell something, and also that a telling agent has to wait for the receiver to listen. In our communication model, things are different. An agent simply receives any message coming in, stores it in its sense buffers (in most other approaches a queue is used for this; in contrast we opt for an unordered buffer), and can decide later what to do with the message. It is possible that an agent totally ignores messages that it received.

The reason we chose asynchronous communication is that it intuitively fits with *agent autonomy* and with the *openness* of many agents. *Autonomy* implies that agents don’t have to react to every message from their environment; they can judge whether reacting is useful for each message coming in. Using synchronous communication, reaction to each message isn’t necessary either, but in case the addressed agent is not willing to listen, the sending agent is suspended. Though there are ways around this (synchronous and asynchronous communication are known to be mutually translatable), this is not matching intuitions around agents. *Openness* means that agents are open to new information and new agents trying to interact with them. So, an open agent never knows what kind of messages to expect. Therefore it would be inconvenient to use  $\text{ask}(\psi, j)$  statements, as the first argument of these statements is the information the agent is interested in. If you don’t know what somebody will have to say to you, there is no appropriate  $\psi$ . These reasons made us opt for asynchronous communication, without *ask*-like statements. Instead, we give each agent a sense buffer, in which all incoming messages are received. The agent can then test this buffer to retrieve the information communicated.

Openness also puts demands on the expressiveness of the content language used for the messages. The fact that agent systems are open and the agents in them are often heterogeneous means that agents are not built in such a way that they “expect” communication on certain issues. Only when agents use a mutually known communication protocol, they know the general flow of the

conversation; otherwise, they don't. This is why the contents of a message needs to have a richer semantics than in classical concurrent programming languages, where processes simply send values around. If someone comes in to your office and says "three", it's impossible to know what she means. But if she instead says "I believe you borrowed three of my books", the matter is clear. The logical language  $\mathcal{L}_{rep(i)}$  offers more expressiveness than the values (numbers, strings, etc.) used in concurrent programming. By using this language, the content of a message can be understandable to the receiver, as the modalities from  $\mathcal{M}_f$  and  $\mathcal{M}_a$  indicate what the status of the argument formula or action is.

As an example of this, assume  $\mathcal{M}_a$  contains *Req-do*, where *Req-do*( $\alpha$ ) means: I request you to do  $\alpha$  for me. If an agent receives the message *Req-do*<sub>*i*</sub>( $\alpha$ ), then it is clear what is asked. Note that the flexibility of the concept sets allows the programmer to introduce speech act types, as used in KQML [46] and in FIPA ACL [133].

The next definition concerns  $\mathcal{A}_{m(i)}$ , the set of basic mental update actions of agent *i*,  $i \in \mathcal{I}$ .

#### DEFINITION 4.8 ( $\mathcal{A}_{m(i)}$ )

The set of basic mental update actions of agent *i* is defined:

$$\mathcal{A}_{m(i)} \equiv \{\text{ins}(C_i(\varphi)), \text{del}(C_i(\varphi)) \mid C \in \mathcal{M}_f, \varphi \in \mathcal{L}_s\} \cup \{\text{ins}(C_i(\alpha)), \text{del}(C_i(\alpha)) \mid C \in \mathcal{M}_a, \alpha \in \text{Interactions}_i\}$$

So, the basic mental update actions in  $\mathcal{A}_{m(i)}$  are adding and removing basic mental formulas to and from its mental state, which is made up from different informational and motivational attitudes. Actually, these are not the only basic mental actions; the agent can also test its own mental state. For technical reasons (regarding the special semantic functions we need to give semantics to the mental update actions), we chose not to include test actions in  $\mathcal{A}_{m(i)}$ .

Next, we come to the set of composite actions which are built only from physical actions and actions of observation and communication, and which can be executed during the interaction phase in a round of the reason–interact cycle:

#### DEFINITION 4.9 (*Interactions*<sub>*i*</sub>)

The set *Interactions*<sub>*i*</sub> of all composite interactions of agent *i* is the smallest set containing:

- \* skip
- \* *a*, where  $a \in \mathcal{A}_s \cup \mathcal{A}_g \cup \mathcal{A}_o \cup \mathcal{A}_{c(i)}$
- \*  $\alpha; \beta$ ,  $\alpha + \beta$ ,  $\alpha \parallel \beta$ , where  $\alpha, \beta \in \text{Interactions}_i$

The basic actions in  $Interactions_i$  are skip and actions from  $\mathcal{A}_s, \mathcal{A}_g, \mathcal{A}_o$  and  $\mathcal{A}_{c(i)}$ . Actions may be composed through the conventional operators ‘;’ for sequential composition, ‘+’ for non-deterministic choice and ‘||’ for parallel composition. During the interaction phase of a reason–interact cycle, agent  $i$  executes composite actions from  $Interactions_i$ . The interaction phase should never last forever; this is why we don’t allow iteration of actions, as an iterated statement could be executed infinitely. Actions can be repeatedly executed by performing them in a number of consecutive reason–interact cycles.

We go on to define the set of composite actions built up only from mental actions. These are the actions an agent can perform during the reasoning phase of the reason–interact cycle.

**DEFINITION 4.10** ( $Intractions_i$ )

The set  $Intractions_i$  of all composite intractions of agent  $i$  is the smallest set containing:

- \*  $\varphi?$ , where  $\varphi \in \mathcal{L}_{rep(i)}$
- \*  $a$ , where  $a \in \mathcal{A}_{m(i)}$
- \*  $\alpha; \beta, \alpha + \beta, \alpha || \beta$ , where  $\alpha, \beta \in Intractions_i$

The basic mental actions an agent can do are testing its mental state and sense buffer, and updating its mental state using the actions from  $\mathcal{A}_{m(i)}$ . These actions may be composed in an arbitrary fashion using sequential composition, non-deterministic choice and parallel composition to obtain the mental part of the agent program. Again, we don’t allow iteration as a program constructor, for the same reasons as we disallowed it for interactions. The mental part of the program is executed during the reasoning phase of the reason–interact cycle.

In an agent program, the agent alternates reasoning and interacting. In the next subsection, we will provide a definition of agent programs that fixes the intuition of the agent executing reason–interact cycles. Before we come to this, we give an example of a partial agent program, that clarifies the kind of internal and external actions an agent executes during one round of the reason–interact cycle.

**EXAMPLE 4.1**

A team of agents are exploring an unknown territory together. Possibly, there are malicious agents around that try to destroy team members. The agents  $i, j$  and  $k$  operate in the territory. The modality set  $\mathcal{M}_f$  contains (among others) an operator  $B$  for belief and an operator  $Q$  for asking questions. The formula  $Q_i(\varphi)$  means that agent  $i$  has the question whether  $\varphi$

holds. Supposing that agent  $i$  is currently moving quite fast, consider the following program code of agent  $i$ :

$$\begin{aligned} & Cd_i(B_j(\text{obstacle\_ahead}))?; \\ & ([B_i(\text{agent\_j\_reliable})?; \text{ins}(B_i(\text{obstacle\_ahead})); \text{EvadeObstacle}] \\ & + \\ & [\neg B_i(\text{agent\_j\_reliable})?; \text{ins}(B_i(Cd_i(B_j(\text{obstacle\_ahead}))))]; \\ & (\text{obs}(\text{obstacle\_ahead}) \parallel \text{comm}(Q_i(\text{obstacle\_ahead}), k) \parallel \text{SlowDown})) \end{aligned}$$

In this program fragment, agent  $i$  reacts in one of two ways when it has received information from  $j$  that there is an obstacle ahead. The two alternative reactions are composed using non-deterministic choice. Both alternatives start with a test of the mental state; this test transforms the non-deterministic choice to a deterministic decision. In case agent  $i$  believes that the bearer of the message ( $j$ , that is) is reliable, it inserts a belief regarding the obstacle in its mental state and subsequently evades the obstacle. In case it doesn't believe that  $j$  is reliable, agent  $i$  is careful; the message about the obstacle could be a trap. Agent  $i$  inserts into its mental state the belief that agent  $j$  communicated to it its belief that there is an obstacle, and then agent  $i$  tries to find out more regarding the alleged obstacle. It concurrently performs an observation and asks agent  $k$  whether it has some information about the obstacle. Precautiously, agent  $i$  also slows down.

### 4.3.3 Syntax of the programming language itself

We need one more auxiliary language, called  $\mathcal{B}_i$ , whose elements are valid *behaviours* (see also Section 4.2.2). A behaviour specifies the body of the reason–interact cycle. In an agent program, one behaviour is iterated, as long as the ultimate objective of the agent is not yet fulfilled.

DEFINITION 4.11 ( $\mathcal{B}_i$ )

$\mathcal{B}_i$  is the smallest set containing:

- \*  $\alpha$ , where  $\alpha \in \text{Interactions}_i$
- \*  $\beta$ , where  $\beta \in \text{Intractions}_i$
- \*  $\alpha; \beta$ , where  $\alpha \in \text{Intractions}_i$  and  $\beta \in \mathcal{B}_i$   
or  $\alpha \in \mathcal{B}_i$  and  $\beta \in \text{Interactions}_i$
- \*  $\alpha + \beta$ , where  $\alpha, \beta \in \mathcal{B}_i$
- \*  $\alpha \parallel \beta$ , where  $\alpha, \beta \in \mathcal{B}_i$

The composite statement making up the behaviour must be constructed in such a way that all reasoning, mental actions precede all physical actions and actions of observation and communication. Therefore, sequential composition of two actions isn't allowed arbitrarily and iteration is not allowed at all. Later on, when we come to the semantics of the programming language, we will take measures to ensure that the execution of a parallel composition of behaviours always results in reasoning preceding acting. And in Section 4.5, we will prove that indeed every execution of a behaviour is proper in this sense.

Although each agent program simply is an iterated behaviour, this doesn't mean that the agent executes the same actions over and over again. Each behaviour still allows many possible execution traces, as it is a composite statement which can contain tests and choices.

For iteration, we use the notation  $\alpha^+$ , meaning: "execute  $\alpha$  one or more times." This isn't the most obvious choice for an iteration operator. Usually, iterating a program  $\alpha$  is done like this:  $\alpha^*$ , which means "execute  $\alpha$  zero or more times". We use  $+$  instead of  $*$  because of a minor technical problem, which we will explain when we define the semantics of agent programs in Section 4.4.2.

At last, we can now define the set of agent programs of the skeleton programming language:

DEFINITION 4.12 (Agent programs)

$\mathcal{S}_i$ , the set of programs for agent  $i$ , is defined:

$$\mathcal{S}_i \equiv \{\varphi? + ((\neg\varphi?; \gamma)^+; \varphi?) \mid \varphi \in \mathcal{L}_{rep(i)}, \gamma \in \mathcal{B}_i\}.$$

We employ the convention to write "while  $\neg\varphi$  do  $\gamma$ " as a syntactically sugared equivalent of  $\varphi? + ((\neg\varphi?; \gamma)^+; \varphi?)$ .

In the definition above,  $\varphi$  is the ultimate objective of the agent. The agent keeps on executing the behaviour  $\gamma$  until  $\varphi$  holds. With this definition, we equip the agents with a syntactic reason–interact cycle.

We give some examples of composed statements and indicate whether they are valid agent programs. We use the convention of writing actions in this font, and *logical formulas in this font*. We will use the while ... do notation for the iterated behaviour, as this is more readable.

#### EXAMPLE 4.2

```
while true do
  ([ $Od_i(window\_dirty)?$ ; ins( $B_i(window\_dirty)$ );
    $B_i(window\_dirty)?$ ; CleanWindow]
  +
  [ $\neg Od_i(window\_dirty)?$ ; skip])
```

This is a (syntactically) valid agent program. Repeatedly, the agent checks whether it observed dirty windows and updates its beliefs accordingly. If there is a dirty window, the agent proceeds to clean it. In this program, each iteration of the cycle first performs some mental actions (updating the beliefs and testing them) and then some physical actions (cleaning windows or doing nothing).

#### EXAMPLE 4.3

```
while true do
  ([ $Od_i(window\_dirty)?$ ; ins( $B_i(window\_dirty)$ );
    $B_i(window\_dirty \wedge \neg at\_window)?$ ; GotoWindow;
    $B_i(window\_dirty)?$ ; CleanWindow]
  +
  [ $\neg Od_i(window\_dirty)?$ ; skip])
```

This program isn't correct, because it allows executions where the agent tests its mental state (which is reasoning) after it performed a physical action of moving. It is very easy to correct the error in the syntax of the previous program, as the second test for a dirty window can just as well be left out.

## 4.4 Semantics

### 4.4.1 Intuitions

The skeleton programming language of which we defined the syntax in the previous section is generic. The system designer can choose the modalities the agents will use, and the reasoning actions as well as the physical actions. In this section, we will provide the generic real-time semantics of the reason–interact cycle based agent systems that can be built in the skeleton programming language.

#### Reasoning and acting

We use two kinds of operational semantics (Plotkin-style, [98]) for the programming language, a local semantics for individual agent programs and a

global semantics which composes the local behaviours of the agents and events happening in the world into a global system behaviour. As the effects of external interactions depends heavily on the behaviour of the other agents and the circumstances in the world, the local semantics results in a large set of candidate traces, of which only a small part actually will contribute to the global system traces that constitute the semantics of the overall system. Only those local agent traces that properly fit together with the local traces of the other agents and fit with the world circumstances will contribute to the global semantics. The other local traces are discarded.

The local semantics focuses on the behaviour of one agent at a time. The configuration of an agent  $i$  is a triple  $\langle \sigma, \delta, \pi \rangle$ , where  $\sigma$  is the sense buffer,  $\delta$  the mental state and  $\pi$  the program still to be executed. Here, both  $\sigma$  and  $\delta$  are sets of formulas;  $\sigma$  is a subset of  $\mathcal{L}_{sb(i)}$  and  $\delta$  is a subset of  $\mathcal{L}_{m(i)}$ . The local semantics yields a set of local traces. Each trace is a sequence of labelled transitions and local configurations, like this:  $S_1 \xrightarrow{l_1} S_2 \xrightarrow{l_2} S_3 \xrightarrow{l_3} S_4$ .

Because of the differences between acting and reasoning, we use two different transition functions for the two phases of the reason–interact cycle. As reasoning is a process internal to an agent, we call the transition function for reasoning the *internal* transition function. Interactions are actions in the environment of the agent, so the transition function for interaction is called the *external* transition function. The reason for this split is the way we deal with parallelism.

Parallelism is allowed almost anywhere in the agent program. But we give a different semantics to parallel operations, depending on whether they are mental actions or physical actions. As the agent is the only one altering its own mental state, it is the responsibility of the agent programmer to use parallel reasoning actions in such a way that they never interfere with one another. An example of a possible interference is when the agent program contains concurrent statements to insert contradictory formulas into the mental state. Assuming the agent programmer prevents situations like this, all parallel actions are independent, so an interleaving semantics is perfectly acceptable for mental actions. On the other hand, the outcome of observations, communications and physical actions an agent performs depends on the actions of the other agents and on events taking place. This is not under total control of the agent. So, we use a step semantics [91] for the interaction part of a reason–interact cycle.

Each internal and external transition step takes one unit of time. This way, we can model that the reasoning process of an agent takes a certain amount of time, and also that physical actions have duration. We assume each atomic mental operation and each action of observing or communication can be exe-

cuted in one time unit. As explained in Section 4.2.3, each individual or group action an agent performs is translated into a sequence of atomic individual sub-actions, each taking one time unit.

The external and internal transition functions use different arcs with different labels. This distinction is necessary to properly define the semantics of our language, because of the dual nature of the semantics needed. For internal transitions, we use  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i} \langle \sigma', \delta', \pi' \rangle$ . The transitions are labelled only with the identity of the agent. One internal transition step denotes the execution of one mental action (either a test or an update). The reasoning phase of one round of the reason–interact cycle generally takes a number of internal transition steps (and thus several time units). Note that the sense buffer can change at an internal transition step. This change isn’t caused by the reasoning of the agent, but by information arriving in the agent from the environment. While the agent is reasoning, other agents can still send messages to it. The reasoning process of the agent can be influenced by this new information. Information coming in during reasoning can make it hard to come to definite and consistent decisions. But as we want our model to be realistic with respect to interaction, we allow these disturbances. For external transitions, we use  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i, (O, C, P)} \langle \sigma', \delta, \pi' \rangle$ . Here,  $i$  again is the agent identifier. The sets  $O$ ,  $C$  and  $P$  contain the observations, outgoing communications and atomic physical sub-action steps the agent  $i$  performs during a particular time unit. These sets must be part of the label of this transition function in order to be able to arrive at the global transition function. Because observing, communicating and physical action differ in nature, we use three distinct sets for them. One external transition step can mean the execution of many physical sub-actions, observations and communications, as in step semantics actions are not interleaved but executed simultaneously in a step-wise manner.

Figure 4.2 illustrates the two different transition functions and the way the reason–interact cycle is interpreted. Every round of the cycle is unfolded into a sequence of mental, internal actions followed by a sequence of sets of atomic external interactions. For the global transition function, the distinction between the two transition functions is not relevant. Using two different transition arrows in the global transition rule would lead to notational clutter, so we replace each internal transition by an appropriate external transition. This will be ex-

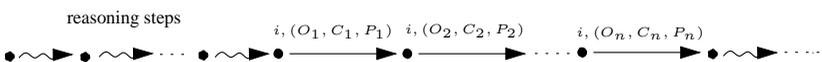


Figure 4.2: Local agent traces

plained in Subsection 4.4.2.

So, the resulting local traces of an agent only use the external transition function. A local trace is a sequence of several ticks of reasoning followed by several ticks of interaction, followed by several ticks of reasoning, etc. The global semantics then selects local traces of the agents that fit together and match the dynamics of the world and composes these into global system traces.

### Regarding the semantics of physical actions

In order to define the transition functions for an agent program, we have to know the most basic actions. The physical actions that occur in agent programs are member of  $A_s$  or  $A_g$ . But actions from these sets can take several ticks of the global clock, and as we explained in Section 4.2.3, we replace physical actions in an agent program by one of its candidate extensions before giving semantics to the program. Each of the candidate extensions is a sequence of atomic sub-actions from  $\mathcal{A}_{sub}$ , which are annotated with signs for success (+) or failure (-). Locally, it cannot be determined whether a sub-action will succeed or fail (as this depends on the world situation), so the local traces will contain both options. The annotation with the sign is important for the nature of the action. In case the environment circumstances are ideal for a certain sub-action, it can only succeed, so only the positively annotated variant of the action will yield successor states for the agent system in the global semantics, while the negatively annotated variant will end in failure, yielding the empty set of successor states. So, the annotated sub-actions  $b_+$  and  $b_-$  are not identical. This is why we define a new set of annotated sub-actions, based on  $\mathcal{A}_{sub}$ . This set is called  $\mathcal{A}_{ex}$ , where *ex* refers to the successful or unsuccessful execution of the sub-actions.

#### DEFINITION 4.13 ( $\mathcal{A}_{ex}$ )

We define the set of sub-actions in execution as follows:

$$\mathcal{A}_{ex} \equiv \{b_+, b_- | b \in \mathcal{A}_{sub} \setminus \{\text{skip}\}\} \cup \{\text{skip}\}$$

As we stated before, we equate skip with skip<sub>+</sub>.

In agent programs, steps of the actions don't appear. Agents decide to do some action, without "being aware of" the atomic sub-actions that make up this action and the time it takes to execute the action. The sub-actions are not allowed to be part of agent programs. But as actions are executed, the (abstract) action is replaced by one of its possible extensions. Depending on the circumstances in the system (the state of the world), one or possibly several extensions will be picked by the semantics and will become part of the set of

execution traces of the agent system. By executing a sequence of sub-actions, each taking one time unit, we implement action duration. Thus, the actions as they occur in agent programs are abstract, as the duration of the execution and (for group actions) the way of realising the action are abstracted from.

The actions of observation and communication, members of  $\mathcal{A}_o$  and  $\mathcal{A}_{c(i)}$ , always take one tick of the global clock. So these actions are not composed of sub-actions; they are atomic themselves.

Below, we define a relation  $\triangleleft$  that links physical actions from  $\mathcal{A}_s$  and  $\mathcal{A}_g$  to their possible extensions. These extensions are sequences of annotated sub-actions (from  $\mathcal{A}_{ex}$ ), linked together by the operator for sequential composition ‘;’. Mathematically, the notation  $\mathcal{A}_{ex}^*$  denotes the set of all sequences of elements of  $\mathcal{A}_{ex}$ . These sequences are not composed by means of sequential composition, but by concatenation, but for the present purpose, we will identify these two forms of building sequences. We also did this (without mentioning it) in Section 4.2.3, when we introduced group action schemes as being multi-sets of sequences of sub-actions.

**DEFINITION 4.14** (Extensions of physical actions)

The relation  $\triangleleft$  (pronounce: “extends into”) is of type  $(\mathcal{A}_s \cup \mathcal{A}_g) \times \mathcal{A}_{ex}^*$ . If the duration of an individual action  $c \in \mathcal{A}_s$  is  $k$ , then there are  $k$  sub-actions  $c_1, c_2, \dots, c_k \in \mathcal{A}_{sub}$ , which can be part of executions of  $c$ . We have:

- $c \triangleleft c_1+; c_2+ \dots; c_k+$
- $c \triangleleft c_1+; c_2+ \dots; c_{k-1}+; c_k-$
- $c \triangleleft c_1+; c_2+ \dots; c_{k-2}+; c_{k-1}-$
- $\vdots$
- $c \triangleleft c_1+; c_2-$
- $c \triangleleft c_1-,$

and for all other sequences  $S$  from  $\mathcal{A}_{ex}^*$  it is not the case that  $c \triangleleft S$ .

If  $A \sqsubseteq \mathcal{A}_{sub}^*$  is a group action scheme for a group action  $d \in \mathcal{A}_g$ , the duration of the group action scheme is  $k$ , and the sequence  $d_1; d_2; \dots; d_k \in A$ , then we have:

- $d \triangleleft d_1+; d_2+ \dots; d_k+$
- $d \triangleleft d_1+; d_2+ \dots; d_{k-1}+; d_k-,$  if  $d_k \neq \text{skip}$
- $d \triangleleft d_1+; d_2+ \dots; d_{k-2}+; d_{k-1}-,$  if  $d_{k-1} \neq \text{skip}$
- $\vdots$
- $d \triangleleft d_1+; d_2-,$  if  $d_2 \neq \text{skip}$
- $d \triangleleft d_1-,$  if  $d_1 \neq \text{skip}.$

There are no other extensions for  $d$  then those resulting from successful or failing executions of contributions from group action schemes for  $d$ .

Note that there can be several group action schemes for a group action, which each contribute a number of extensions of the group action. In Section 4.2.3, we gave an example of the generation of extensions of a group action (the LiftTable example).

In the semantics of the skeleton programming language, we will use the relation  $\triangleleft$  to replace physical actions in local agent programs by one of their extensions. The local semantics of an agent program yields a set of traces for the execution of the program of one agent. In this local semantics, each extension is considered a possibility, and traces are generated for each of the possible extensions. We have to generate all of the local traces, because the local agent state is not sufficient to determine which local behaviour will be executed. It is essential for the local semantics that the *first* sub-action of an extension is recognisable as such, because the execution of a physical action can be preceded by an arbitrary delay. When we come to the semantics of physical actions, this will become clear. At this point, we assume that the first sub-action of an extension is always subscripted with a 1 (although we are sloppy with this in the examples).

In the global semantics, the local agent behaviours and the dynamics of the physical world (events) are combined into system traces. The global semantics takes a local trace of each agent in the system and, taking into account events happening in the world, intertwines these into a system trace. In constructing system traces, only local agent traces matching the aptness of the world situation for execution of actions will be used.

## 4.4.2 Local semantics: agent traces

In this subsection, we will give operational semantics for the local program of one agent. In order to do this, we first need to formally define local agent configurations, which are transformed by steps of the transition function. We also need to define extensions of the sets of agent programs, as partial execution of a syntactically correct program does not necessarily result in another syntactically correct program. To be precise, if  $\pi$  is a program according to Definition 4.12 and  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i} \langle \sigma', \delta', \pi' \rangle$  or  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i, (O, C, P)} \langle \sigma', \delta, \pi' \rangle$  is an execution step of this program, then the remaining composite statement  $\pi'$  need not be a program. We go into this soon.

### Some definitions

As we sketched in Section 4.4.1, an agent configuration consists of three elements: the sense buffer, the mental state and the program remaining to be exe-

cutted. This last configuration element is the cause of a small technical problem, which prohibits a straightforward definition of local configurations.

Earlier, in Definition 4.12, we defined the syntax of well-formed agent programs. Each agent program (as written by the programmer) has to be an iterated composite behaviour. The set  $\mathcal{S}_i$  contains all programs of agent  $i$ . But when an agent program from  $\mathcal{S}_i$  is executed, and some statements of the programs are finished, the remainder of the program left to be executed is not very likely to be another element of  $\mathcal{S}_i$ . One of the things that happens when interpreting agent programs is that abstract physical actions are textually replaced by one of their extensions, and the definition of the syntax of programs doesn't allow sub-actions as part of programs. Also, execution of the iterated composite statement which is the agent program generally doesn't result in another iterated composite statement, as the transition rule for iteration given below will show. For this reason, we can't straightforwardly define the third element of a local configuration of agent  $i$  to be an element of  $\mathcal{S}_i$ .

We need extensions of the sets  $\mathcal{S}_i$ . The set  $\mathcal{S}_i^-$  contains all *program remainders* resulting from one or more execution steps of programs from  $\mathcal{S}_i$ .

Local agent configurations, program remainder sets and transition rules are defined by mutual recursion. We first give the formal definition of local agent configurations:

**DEFINITION 4.15 (Local agent configuration)**

The local configuration of an agent  $i$  is a triple  $\langle \sigma, \delta, \pi \rangle$ , where

- \*  $\sigma \subseteq \mathcal{L}_{sb(i)}$  is the sense buffer
- \*  $\delta \subseteq \mathcal{L}_{m(i)}$  is the private mental state
- \*  $\pi \in \mathcal{S}_i^-$  is the program to be executed

Next, we define the remainder sets, using the transition functions we will define below by means of the transition system. Recall that we use  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i}$   $\langle \sigma', \delta', \pi' \rangle$  for internal transitions, and  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i, (O_i, C_i, P)}$   $\langle \sigma', \delta, \pi' \rangle$  for external transitions.

**DEFINITION 4.16 (Program remainders)**

The set  $\mathcal{S}_i^-$  of remainders of programs of agent  $i$  is inductively defined as the smallest set satisfying:

- \*  $\surd \in \mathcal{S}_i^-$
- \*  $\mathcal{S}_i \subseteq \mathcal{S}_i^-$
- \* If  $\pi \in \mathcal{S}_i^-$ , and either  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i} \langle \sigma', \delta', \pi' \rangle$  for some  $\sigma, \delta, \sigma'$  and  $\delta'$  or  $\langle \sigma, \delta, \pi \rangle \xrightarrow{i, (O, C, P)} \langle \sigma', \delta, \pi' \rangle$  for some  $O, C, P, \sigma, \sigma'$  and  $\delta$ , then  $\pi' \in \mathcal{S}_i^-$ .

We use the symbol  $\surd$  to denote the empty program remainder; this results if there are no more statements left to be executed. We assume  $\pi \parallel \surd = \surd \parallel \pi = \pi$  and also  $\surd; \pi = \pi$ . The set of program remainders for agent  $i$  thus contains all programs of agent  $i$ , and the composite statements resulting from one or more execution steps of an agent program of  $i$ .

**Transition rules**

During each atomic transition step, the sense buffer can change because of observation results or communication coming in. Although the arrival of observation results is expected, the contents of the results is determined by the state of the world. Incoming communication isn't under control of the agent at all. So, locally we can't specify the changes to the sense buffer. Therefore, we allow the sense buffer to change in an arbitrary fashion during each atomic transition step. Arbitrary changes are also used in the semantics of communication in CSP [61]. The global semantics will select the appropriate change(s) to the sense buffer, taking into account the messages other agents sent to the agent and the result of observational actions the agent performed. We allow changes to the sense buffer during both the reasoning phase and the interaction phase. Especially new information coming in during the reasoning part of the reason–interact cycle can have disrupting effects, as the agent is using information from the sense buffer in order to derive conclusions during reasoning. If the sense buffer isn't stable, it may be hard or impossible for the agent to come to definite conclusions.

We now give semantics to the program of an agent with identifier  $i \in \mathcal{I}$ , starting with the atomic external actions. This is the transition rule for the action of doing nothing, skip:

$$\frac{}{\langle \sigma, \delta, \text{skip} \rangle \xrightarrow{i, (\emptyset, \emptyset, \emptyset)} \langle \sigma', \delta, \surd \rangle}$$

This transition for the skip action is a physical transition, as `skip` is a member of  $Interactions_i$ . Recall that the label of a transition  $i, (O, C, P)$  is interpreted like this:  $i$  is the identifier of the agent performing the action,  $O$  is the set of observation actions performed during this computation step,  $C$  is the set of communicative actions performed and  $P$  is the set of atomic physical sub-actions performed. As `skip` means doing nothing, all three sets in the resulting transition are empty.

Note that the sense buffer  $\sigma$  is transformed into  $\sigma'$ , without any specification of the relation between  $\sigma$  and  $\sigma'$ . This means that the sense buffer can change in an arbitrary fashion locally.

Next, we look at physical actions. Actions in  $A_s$  and  $A_g$  actually stand for sequences of annotated atomic sub-actions from  $\mathcal{A}_{ex}^*$ . The sub-actions always are single-agent actions taking one time unit to execute. In Definition 4.14, we introduced the relation  $\triangleleft$ ; if  $a$  is a physical action, and  $S$  is a sequence of sub-actions, then  $a \triangleleft S$  means that  $S$  is a possible extension (execution sequence) of  $a$ . We assume that before giving semantics to a physical action, it extends into one of the possible sequences. If there are  $n$  possible extensions, this leads to  $n$  program variants. Thus, in the transition rules given below we only have to interpret atomic single-agent sub-actions<sup>3</sup>.

Before we show the transition rules for action execution, we compare our practice of representing action duration using extensions with research on the formal semantics of concurrent processes. Here, actions with duration are often represented by a pair of events (where an *event* is an instantaneous action, in contrast to our use of the term), the first denoting the start of the action and the second denoting its finish ([62], page 24). So, each action is replaced by two sub-actions, one denoting the start of the action and the other the finish. Next, an ordinary interleaving semantics is used. In an article by Rob van Glabbeek and Frits Vaandrager [52], this splitting of actions is generalised. Actions can be split into any number of sub-actions, which is close to what we do. But the authors show that if an interleaving semantics is used:

‘... it is impossible to capture durational or structural properties of actions by splitting them into sequences of any finite number of parts.’ (page 3)

This advocates our use of a semantics which is non-interleaving, but inspired by step semantics ([91]).

---

<sup>3</sup>One might wonder why we didn’t handle action expansion in a transition rule. The reason for this is that each transition step, either internal or external, takes one unit of time. Action expansion isn’t something the agent performs; being merely a technical operation, it shouldn’t take any time. So, it is dealt with outside the transition system.

For the following transition rules, we assume  $a \in \mathcal{A}_s \cup \mathcal{A}_g, a \triangleleft a_1; a_2; \dots; a_n$  and  $1 \leq r \leq n$ . Recall from Definition 4.14 that  $a_1, a_2, \dots, a_n$  are annotated with  $+$  or  $-$ , to indicate a successful or failing sub-action.

$$\frac{}{\langle \sigma, \delta, a_1 \rangle \xrightarrow{i, (\emptyset, \emptyset, \emptyset)} \langle \sigma', \delta, a_1 \rangle}$$

$$\frac{}{\langle \sigma, \delta, a_r \rangle \xrightarrow{i, (\emptyset, \emptyset, \{a_r\})} \langle \sigma', \delta, \surd \rangle}$$

There are a number of things to be noted here:

- \* There can be an arbitrarily long delay before the first atomic sub-action of a physical action is executed. The first transition rule above implements this idling; the agent doesn't act yet. This transition rule only applies to the first sub-action of an extension, which is why we demanded that these first sub-actions are clearly recognisable (being subscripted with a 1). As soon as the first sub-action has been done, no delays are possible anymore.
- \* The mental state doesn't change as a consequence of performing an atomic physical action. This is because these actions are external actions, that depend on and directly affect the environment of the agent. So, we entirely separate physical actions from mental actions. The effects of physical actions on the world cannot be given at the local agent level, as the world is not part of the local agent state.
- \* The consecutive atomic sub-actions of physical actions are sequenced simply by the operator  $;$ , which will be given the usual semantics later on.

Next, we interpret observation and communication. Suppose  $o \in \mathcal{A}_o$ . Then:

$$\frac{}{\langle \sigma, \delta, o \rangle \xrightarrow{i, (\emptyset, \emptyset, \emptyset)} \langle \sigma', \delta, o \rangle}$$

$$\frac{}{\langle \sigma, \delta, o \rangle \xrightarrow{i, (\{o\}, \emptyset, \emptyset)} \langle \sigma', \delta, \surd \rangle}$$

Observative actions are also preceded by an arbitrarily long delay, which is what the first transition rule provides. At the local agent level, we don't know

what the outcome of the observation might be, as this depends on the environment. So, the sense buffer changes in an arbitrary manner, as in all previous transition rules.

Next, we look at actions of communicating something. Suppose  $c \in \mathcal{A}_{c(i)}$ .

$$\frac{}{\langle \sigma, \delta, c \rangle \xrightarrow{i, (\emptyset, \emptyset, \emptyset)} \langle \sigma', \delta, c \rangle}$$

$$\frac{}{\langle \sigma, \delta, c \rangle \xrightarrow{i, (\emptyset, \{c\}, \emptyset)} \langle \sigma', \delta, \sqrt{} \rangle}$$

Again, the first rule provides a delay before the communicative action.

Now, we have looked at all building blocks of interactions. Next, we turn to the basic actions forming the building blocks of intractions, that is, to basic mental actions. We use our other transition function ( $\overset{i}{\rightsquigarrow}$ ) for these. We start with testing the mental state and sense buffer:

$$\frac{\delta \cup \sigma \models \varphi}{\langle \sigma, \delta, \varphi? \rangle \overset{i}{\rightsquigarrow} \langle \sigma', \delta, \sqrt{} \rangle}$$

The formula  $\varphi$  is tested on the local agent state, which consists of the sense buffer and the mental state. In case  $\varphi$  is true, the transition can be taken; otherwise, execution blocks.

The other basic mental actions are actions from  $\mathcal{A}_{m(i)}$ . These actions update the mental state by adding or deleting intentions, beliefs, desires, goals, etc. To do this in a logically justified manner, we introduce a *revision function for the mental state of agent  $i$* ,  $v_i : \wp(\mathcal{L}_{m(i)}) \times \mathcal{A}_{m(i)} \rightarrow \wp(\wp(\mathcal{L}_{m(i)}))$ . This function takes a mental state and a mental action and returns the set of mental states that could result from a revision of the mental state caused by the action. We abstract from details of revision; see e.g., [49] for an overview of techniques for belief revision. Then, this is the transition rule for a mental action  $m \in \mathcal{A}_{m(i)}$ , supposing  $\delta' \in v_i(\delta, m)$ :

$$\frac{}{\langle \sigma, \delta, m \rangle \overset{i}{\rightsquigarrow} \langle \sigma', \delta', \sqrt{} \rangle}$$

This transition rule clearly illustrates that our semantics is a *generic* semantics for agent architectures. As the system designer can choose which modalities and concepts (belief, expectation, desire, request, intention, goal, ...) the agent

will employ, the generic semantics is parameterised with a revision function for the modalities picked.

Now we will give semantics to the program constructors. Because we have two transition functions now, one for mental actions and one for physical actions, this is not entirely traditional. We want to make sure that all reasoning actions are finished the moment the interaction phase of a reason–interact cycle starts. This influences the semantics of the program constructors. We will use the arrow  $\dashrightarrow^l$  to stand for either  $\xrightarrow{l}$  or  $\rightsquigarrow^l$ , where  $l$  denotes an arbitrary label (possibly composed of several elements).

We start with sequential composition. Nothing strange happens here.

$$\frac{\langle \sigma, \delta, \pi_1 \rangle \dashrightarrow^l \langle \sigma', \delta', \pi'_1 \rangle}{\langle \sigma, \delta, \pi_1; \pi_2 \rangle \dashrightarrow^l \langle \sigma', \delta', \pi'_1; \pi_2 \rangle}$$

This transition rule states that in order to execute a sequential composition of two programs, you start by executing the first step of the first program.

We look at non-deterministic choice next. According to the syntax of the programming language, the operator  $+$  can occur between two elements of  $\mathcal{B}_i$ , two elements of  $Interactions_i$  and two elements of  $Intractions_i$ . The first case appears interesting, as one of the behaviours could start off doing an external interaction step, while the other starts with a reasoning step. But the syntax of the programming language is defined in such a manner that either behaviour can be chosen, without running the risk of reasoning after acting during one round of the reason–interact cycle. Later on, in Section 4.5, we will prove this.

$$\frac{\langle \sigma, \delta, \pi_1 \rangle \dashrightarrow^l \langle \sigma', \delta', \pi'_1 \rangle}{\langle \sigma, \delta, \pi_1 + \pi_2 \rangle \dashrightarrow^l \langle \sigma', \delta', \pi'_1 \rangle}$$

$$\frac{\langle \sigma, \delta, \pi_2 \rangle \dashrightarrow^l \langle \sigma', \delta', \pi'_2 \rangle}{\langle \sigma, \delta, \pi_1 + \pi_2 \rangle \dashrightarrow^l \langle \sigma', \delta', \pi'_2 \rangle}$$

Next, we look at iteration. An iterated program  $\pi^+$  is executed either one or more times. For both cases, we have a transition rule:

$$\frac{\langle \sigma, \delta, \pi \rangle \dashrightarrow^l \langle \sigma', \delta', \pi' \rangle}{\langle \sigma, \delta, \pi^+ \rangle \dashrightarrow^l \langle \sigma', \delta', \pi' \rangle}$$

$$\frac{\langle \sigma, \delta, \pi \rangle \xrightarrow{l} \langle \sigma', \delta', \pi' \rangle}{\langle \sigma, \delta, \pi^+ \rangle \xrightarrow{l} \langle \sigma', \delta', \pi'; \pi^+ \rangle}$$

After the first step of the program  $\pi$  is taken, and the remaining program to be executed is  $\pi'$ , the first rule replaces all of the iterated statement  $\pi^+$  by  $\pi'$ . This way, the program  $\pi$  is executed one time only. The second transition rule results in a remaining program in which first  $\pi'$  is finished, after which there again will be one or more executions of  $\pi$ . So, the second transition rule results in two or more executions of  $\pi$ .

At this point we can explain why we didn't choose for the usual iteration operator  $**$ . The statement  $\pi^*$  means that  $\pi$  is executed either zero or more times. A transition rule for the first case would have to look like this:

$$\frac{}{\langle \sigma, \delta, \pi^* \rangle \xrightarrow{i} \langle \sigma', \delta', \surd \rangle}$$

Now, there are two objections to this rule. The first is that it is an arbitrary choice to use an internal transition step here. We could just as well use an external transition step, labelled like this:  $\xrightarrow{i, (\emptyset, \emptyset, \emptyset)}$ . As nothing is happening in the transition, it is impossible to classify this step as either external or internal. Secondly, there is a problem regarding time. Each transition, either internal or external, takes one tick of the clock. So, zero executions of an iterated statement will take a time unit, even though nothing is actually executed. These two problems made us opt for the  $^+$  iteration operator instead of the  $**$ .

Now, we come to the transition rules for the local program constructor of parallel composition. Parallel composition in particular is interesting because we want to implement simultaneousness in the semantics of parallel executions. Because we have two transition functions now, one for mental actions and one for physical actions, the transition rules are rather subtle. When there are two program fragments in parallel, there are several cases. When the first execution steps of both fragments are internal steps, they are interleaved. When the first steps of both fragments are external, they are taken simultaneously, such that their sets of atomic actions are unified. But when one fragment takes an internal step, and the other an external step, then we give precedence to the internal step. This is because we want the reasoning phase of a round of the reason–interact phase to be totally finished before the interaction phase is started.

These are the transition rules:

$$\frac{\langle \sigma, \delta, \pi_1 \rangle \xrightarrow{i, (O_1, C_1, P_1)} \langle \sigma', \delta, \pi'_1 \rangle, \langle \sigma, \delta, \pi_2 \rangle \xrightarrow{i, (O_2, C_2, P_2)} \langle \sigma', \delta, \pi'_2 \rangle}{\langle \sigma, \delta, \pi_1 \parallel \pi_2 \rangle \xrightarrow{i, (O_1 \cup O_2, C_1 \cup C_2, P_1 \cup P_2)} \langle \sigma', \delta, \pi'_1 \parallel \pi'_2 \rangle}$$

$$\frac{\langle \sigma, \delta, \pi_1 \rangle \xrightarrow{i} \langle \sigma', \delta', \pi'_1 \rangle}{\langle \sigma, \delta, \pi_1 \parallel \pi_2 \rangle \xrightarrow{i} \langle \sigma', \delta', \pi'_1 \parallel \pi_2 \rangle} \quad \frac{\langle \sigma, \delta, \pi_2 \rangle \xrightarrow{i} \langle \sigma', \delta', \pi'_2 \rangle}{\langle \sigma, \delta, \pi_1 \parallel \pi_2 \rangle \xrightarrow{i} \langle \sigma', \delta', \pi_1 \parallel \pi'_2 \rangle}$$

We will explain the way these rules work by means of an example. The example illustrates that in parallel compositions, the internal steps of the constituent programs will precede the external interaction steps.

#### EXAMPLE 4.4 (Three parallel programs)

Suppose we have three composite statements in parallel, that is, the program to be executed is  $\pi_1 \parallel \pi_2 \parallel \pi_3$ . We will show how three traces, of  $\pi_1$ ,  $\pi_2$  and  $\pi_3$  are combined into a trace of the parallel composition. We abstract from the changes to the mental state and the sense buffer, as these are not relevant for the present matter. We will thus look at abstract transitions of the form  $\pi \xrightarrow{l} \pi'$  instead of  $\langle \sigma, \delta, \pi \rangle \xrightarrow{l} \langle \sigma', \delta', \pi' \rangle$ . The trace of  $\pi_1$  consists of two reasoning steps, followed by one interaction step:  $\pi_1 \xrightarrow{i} \pi'_1 \xrightarrow{i} \pi''_1 \xrightarrow{i, (O_1, C_1, P_1)} \checkmark$ . The trace of  $\pi_2$  only has external steps:  $\pi_2 \xrightarrow{i, (O_2, C_2, P_2)} \pi'_2 \xrightarrow{i, (O_3, C_3, P_3)} \checkmark$ . Finally, the trace of  $\pi_3$  has one internal and one external step:  $\pi_3 \xrightarrow{i} \pi'_3 \xrightarrow{i, (O_4, C_4, P_4)} \checkmark$ . To compute the execution trace of the parallel composition, we have to put parentheses into the parallel composition (it doesn't matter where, because parallel composition is associative, as we will justify in Section 4.5). So, we compute the semantics of  $(\pi_1 \parallel \pi_2) \parallel \pi_3$ . Because the first step of  $\pi_1$  is  $\pi_1 \xrightarrow{i} \pi'_1$  and the first step of  $\pi_2$  is  $\pi_2 \xrightarrow{i, (O_2, C_2, P_2)} \pi'_2$ , only the second transition rule above applies, resulting in  $\pi_1 \parallel \pi_2 \xrightarrow{i} \pi'_1 \parallel \pi_2$ . This transition is interleaved with the first internal transition step of  $\pi_3$ , where the order is picked randomly. So, one of the two possibilities is a trace starting like this:  $(\pi_1 \parallel \pi_2) \parallel \pi_3 \xrightarrow{i} (\pi_1 \parallel \pi_2) \parallel \pi'_3 \xrightarrow{i} (\pi'_1 \parallel \pi_2) \parallel \pi'_3$ . At this point, the next step of  $\pi'_1$  is its second internal step, while the next steps of both  $\pi_2$  and  $\pi'_3$  are external transitions. According to the rules above, the only possibility is that  $\pi'_1$  proceeds:

$(\pi_1 \parallel \pi_2) \parallel \pi_3 \xrightarrow{i} (\pi_1 \parallel \pi_2) \parallel \pi'_3 \xrightarrow{i} (\pi'_1 \parallel \pi_2) \parallel \pi'_3 \xrightarrow{i} (\pi'_1 \parallel \pi_2) \parallel \pi'_3$ . Now, all reasoning steps have been taken and the interactions can be executed. First, we look at the sub-program  $(\pi'_1 \parallel \pi_2)$ . As we have  $\pi'_1 \xrightarrow{i, (O_1, C_1, P_1)} \checkmark$  and  $\pi_2 \xrightarrow{i, (O_2, C_2, P_2)} \pi'_2$ , the first transition rule allows us to derive that  $\pi'_1 \parallel \pi_2 \xrightarrow{i, (O_1 \cup O_2, C_1 \cup C_2, P_1 \cup P_2)} \checkmark \parallel \pi'_2$ . As  $\checkmark \parallel \pi'_2 = \pi'_2$ , and  $\pi'_3$  also takes an external step, we get  $(\pi'_1 \parallel \pi_2) \parallel \pi'_3 \xrightarrow{i, (O_1 \cup O_2 \cup O_4, C_1 \cup C_2 \cup C_4, P_1 \cup P_2 \cup P_4)} \pi'_2 \xrightarrow{i, (O_3, C_3, P_3)} \checkmark$ , which is the complete trace of this parallel composition.

In the following, we will sometimes use the notations  $S \not\rightsquigarrow S'$  and  $S \not\rightarrow S'$  to denote that there is no internal or external transition step possible, respectively.

## Resulting local traces

The local transition rules result in local traces which look like this:

$$\begin{aligned}
 & (\sigma_0, \delta_0, \pi_0) \xrightarrow{i} (\sigma_1, \delta_1, \pi_1) \xrightarrow{i} (\sigma_2, \delta_2, \pi_2) \xrightarrow{i} \dots \xrightarrow{i} (\sigma_n, \delta_n, \pi_n) \xrightarrow{l_1} \\
 & (\sigma_{n+1}, \delta_n, \pi_{n+1}) \xrightarrow{l_2} (\sigma_{n+2}, \delta_n, \pi_{n+2}) \xrightarrow{l_3} \dots \xrightarrow{l_m} (\sigma_{n+m}, \delta_n, \pi'_0) \xrightarrow{i} \\
 & \dots
 \end{aligned}$$

So, every trace is a finite or infinite alternation of reasoning steps and interaction steps. Note again the separation between internal and external agent behaviour. During the reasoning phase, the mental state ( $\delta$ ) changes. During the interaction phase, it remains the same, as performing physical actions, observations and communications never directly affect the mental state of the agent. Each transition, either internal or external, takes one time unit, making our semantics a real-time semantics. Although not detectable in the local traces, an agent reasoning doesn't directly influence the state of the world or the mental state of the other agents. Agents in their interaction phase do modify the world state and the sense buffers.

In order to combine the processing of all agents into one system trace, it is convenient to only have one kind of transition arrows. So, after we generated the set of local traces for an agent program, we replace every internal transition  $(\sigma, \delta, \pi) \xrightarrow{i} (\sigma', \delta', \pi')$  in a local agent trace with an external transition  $(\sigma, \delta, \pi) \xrightarrow{i, (\emptyset, \emptyset, \emptyset)} (\sigma', \delta', \pi')$ . The three empty sets labelling the transition indicate that no interactions are done during this reasoning step.

### 4.4.3 Global semantics: system traces

On the global level, we link up the local traces of all agents in the multi-agent system. Also, we allow events to happen in the world. As the local reasoning steps (the intractions) of the different agents are independent of the processing of the other agents or the world situation, the global semantics doesn't affect the locally computed semantics of intractions. But the effects of locally computed interaction steps of the agents do depend on each other and the events taking place in the world. So, the main use of the global semantics is to compute the result of individual physical actions, observation, communication and events as they take place concurrently in possibly interfering or synergetic ways.

In order to define the global semantics, we first need definitions of the state of the world and of the overall system configuration.

#### Some definitions

First, we define the state of the world simply to be the set of propositional atoms currently true of the environment:

DEFINITION 4.17 (World state)

A world state  $W$  is an interpretation of the set of propositional atoms  $\mathcal{P}$ , defined as the set of atoms that are *true*:  $W \subseteq \mathcal{P}$ . All atoms not part of a world state are *false*.

Next we define *global system configurations*:

DEFINITION 4.18 (Global system configuration)

We define a global system configuration to be a pair  $\langle W, \{S_i \mid i \in \mathcal{I}\} \rangle$ , where  $W \subseteq \mathcal{P}$  is the world state, and the  $S_i$ 's are local agent configurations.

#### The global transition rule

Because the local behaviour of a single agent is now placed in the context of the behaviours of the other agents and the world, there are several matters that need attention at this level. These are:

- \* Updating of the sense buffers of the agents. The sense buffer of an agent is *not* updated through deliberation of the agent, but only through observation results and messages coming in. To update the sense buffers, we introduce functions  $\rho_i, i \in \mathcal{I}$ , which takes a sense buffer of agent  $i$  and a

set of new formulas (incoming observations and communication) for the sense buffer, and returns a set of updated sense buffers. The outcome is a set because the revision process might yield several new sense buffers. One of these is non-deterministically chosen. It is not enough to just add the formulas to the old sense buffer, as new information could contradict older information. Additionally, we need a function for observability. We use the functions  $\xi_i, i \in \mathcal{I}, \xi_i : \wp(\mathcal{P}) \times \mathcal{L}_w \rightarrow \{0, 1\}$ . This function takes the world state and the formula which is being observed, and returns 1 when the formula is observable in that particular world state and 0 otherwise.

- \* Events taking place in the physical world. As events are occurrences not initiated by any of the agents in the system, we introduce them at the global level.
- \* Changing the world state. Performing actions probably will change the state of the world. We have a function  $\tau$ , which takes a world state, and a pair consisting of events and annotated atomic single action steps with the identifiers of the agents performing them, and then returns a set of possible new world states. We have a set of world states as it might not be totally predictable what the outcome of a combination of actions and events will be. The set of world states represents the possible outcomes, of which one is non-deterministically chosen.

Note that the function  $\tau$ , which models world state transformation, incorporates all details about which actions and events interfere with each other, and in which way. In the next subsection, we will touch upon some aspects of  $\tau$ , such as the role it plays in modelling group actions.

As introduced in Section 4.3.1,  $\mathcal{E}$  is the set of all events possible in the world. For each event  $e \in \mathcal{E}$ , there is a precondition  $\varphi_e \in \mathcal{L}_w$ , which expresses in which world states the event could take place. If this condition isn't fulfilled in a world state, the event can't take place in that state. For example, the event GasExplosion can only take place in case someone forgot to turn off the gas or there is a gas leak. If this is the case, it is not certain that there will be a GasExplosion, but it is possible. In the following,  $D \subseteq \mathcal{E}$  is the set of events taking place in some time unit,  $S_i$  abbreviates  $\langle \sigma_i, \delta_i, \pi_i \rangle$  and  $S'_i$  abbreviates  $\langle \sigma'_i, \delta'_i, \pi'_i \rangle$ .

Now, we have the following global transition rule:

$$\frac{\forall i \in \mathcal{I} : S_i \xrightarrow{i, (O_i, C_i, P_i)} S'_i, \quad W \models \{\varphi_e \mid e \in D\}}{\langle W, \{S_i \mid i \in \mathcal{I}\} \rangle \longrightarrow \langle W', \{S'_i \mid i \in \mathcal{I}\} \rangle}$$

where the following must hold:

\* For all  $i \in \mathcal{I}$  :

$$\begin{aligned} \sigma'_i \in \rho_i(\sigma_i, & \\ & \{Od_i(\varphi) \mid \text{obs}(\varphi) \in O_i \text{ and } \xi_i(W, \varphi) = 1 \text{ and } W \models \varphi\} \cup \\ & \{-Od_i(\varphi) \mid \text{obs}(\varphi) \in O_i \text{ and } (W \not\models \varphi \text{ or } \xi_i(W, \varphi) = 0)\} \cup \\ & \{Cd_i(\varphi) \mid \text{exists } j \neq i \in \mathcal{I} : \text{comm}(\varphi, i) \in C_j\}). \end{aligned}$$

This states that all sense buffers must have been properly updated.

\*  $W' \in \tau(W, (D, \{(a, i) \mid i \in \mathcal{I} \text{ and } a \in P_i\}))$ . This computes the next world state.

This transition rule yields the global execution traces of a multi-agent system specified in our programming language. Intuitively, the global transition sums up all actions and events taking place during one unit of time. We will explain the details of the rule and the conditions that belong to it.

The body of the rule (the rule without the prerequisites and conditions) actually doesn't seem to do anything spectacular. It simply takes a local transition step of each agent, and integrates all local state changes, from  $S_i$  to  $S'_i$ , into a global state change  $\langle W, \{S_i \mid i \in \mathcal{I}\} \rangle \longrightarrow \langle W', \{S'_i \mid i \in \mathcal{I}\} \rangle$ . A local transition step of an agent models a set of observative, communicative and physical sub-actions being executed during one unit of time. Together with the events taking place during this time step,  $D$ , these sub-actions determine the state change of the world, from  $W$  to  $W'$ .

But not every combination of local transition steps will result in a global transition. This is due to the two conditions of the transition rule. We start by looking at the first condition and its selective function.

Recall that in the local transition rules, we allowed arbitrary changes to the sense buffer in each local transition. This results in many transitions which are clearly wrong. For example, the sense buffer can change from the empty set to a set of observation results completely specifying the observable world state, without the agent executing any observations. Also, when the agent is performing an observation regarding some formula, the sense buffer can contain an observation result completely opposite to the actual world state, and an arbitrary set of messages, which might not have been sent at all by the other agents. The first condition above filters out these wrong local transitions. Only

if the change to the sense buffer as specified in the local transition matches the observations the agent is actually performing and the results as depending on the current state of the world ( $W$  in the rule above), as well as the messages the other agents actually send to this agent in this clock tick, the local transition could contribute to the global transition.

We look at the details of the first condition. It states that for each of the agents in the multi-agent system ( $i \in \mathcal{I}$ ), the new sense buffer as produced in the local transition ( $\sigma'_i$ ) must be a revised version of the original sense buffer ( $\sigma'_i \in \rho_i(\sigma_i, \dots)$ ). The sense buffer must be revised with a set of formulas, consisting of three subsets:

- \*  $\{Od_i(\varphi) | \text{obs}(\varphi) \in O_i \text{ and } \xi_i(W, \varphi) = 1 \text{ and } W \models \varphi\}$ . These are the positive observation results (“It is observed that  $\varphi$  holds”) of observative actions agent  $i$  performed in this clock tick ( $\text{obs}(\varphi) \in O_i$ ), where the results are positive because the formulas involved are observable to  $i$  in this world state ( $\xi(W, \varphi) = 1$ ) and also hold ( $W \models \varphi$ ).
- \*  $\{-Od_i(\varphi) | \text{obs}(\varphi) \in O_i \text{ and } (W \not\models \varphi \text{ or } \xi_i(W, \varphi) = 0)\}$ . These are the negative observation results (“It is *not* observed that  $\varphi$  holds”) of observative actions agent  $i$  performed in this clock tick ( $\text{obs}(\varphi) \in O_i$ ), where the results are negative because the formulas involved are not observable to  $i$  in this world state ( $\xi(W, \varphi) = 0$ ) or do not hold ( $W \not\models \varphi$ ). Note that a negative observation result  $\neg Od_i(\varphi)$  is very different from  $Od_i(\neg\varphi)$ !
- \*  $\{Cd_i(\varphi) | \text{exists } j \neq i \in \mathcal{I} : \text{comm}(\varphi, i) \in C_j\}$ . These are the results (“It is communicated that  $\varphi$  holds”) of other agents ( $j \neq i \in \mathcal{I}$ ) communicating something to agent  $i$  ( $\text{comm}(\varphi, i) \in C_j$ ).

Only if the update of the sense buffer in a local transition obeys the first condition, the local transition could become part of the global computation step. But in order for this to happen, the second condition, regarding the next world state, also has to hold. It might not be immediately clear how this condition rules out certain local agent transitions, as the condition only prescribes the change to the world state. The condition states that the new world state ( $W'$ ) is a result of the occurrence of a set of events ( $D$ ) and of the performing of a set of atomic single-agent sub-actions by certain agents ( $\{(a, i) | i \in \mathcal{I} \text{ and } a \in P_i\}$ ) in the old world state ( $W$ ). Recall that these sub-actions are annotated with either a + for success or - for failure. Note that the sub-actions in the argument of  $\tau$  are paired up with the identity of the agent doing them; this is so because the effect of performing a sub-action might depend on which agent executes it. The function  $\tau$  takes into account the interference or synergy (in case of group actions) of the atomic actions, and yields a set of new world states.

But in case the given combination of sub-actions or events *cannot* be executed, the result of  $\tau$  will be the empty set, and the local transitions giving rise to the sub-actions won't contribute to the global system execution. So, the function  $\tau$  determines whether an annotated sub-action can be executed in certain environment circumstances (world situation, other sub-actions and events taking place). For example, if the local transition of a certain agent  $i$  contains a successful sub-action, that is,  $\mathbf{a}_{k+} \in P_i$  for  $\mathbf{a}_k \in \mathcal{A}_{sub}$ , and the circumstances in the world  $W$  are such that this sub-action can only fail, then  $\tau(W, (D, A))$  will yield the empty set of successor world states for each  $A$  containing  $(\mathbf{a}_{k+}, i)$ . Thus, no global transition is possible in which  $\mathbf{a}_k$  succeeds. But the local semantics also provides local traces of agent  $i$  containing the failing variant of this sub-action, that is,  $\mathbf{a}_{k-} \in P_i$ . For this  $P_i$  and the same world state  $W$ ,  $\tau(W, (D, A))$  will yield a set of successor states, representing the world states that can result from the failing sub-action in combination with the other occurrences taking place. As another example, suppose two agents are executing interfering sub-actions, which never can be successfully executed together. Then, only negatively annotated variants of the sub-actions will lead to a new world state using  $\tau$ . Sometimes, there are more options than one. For example, if three sub-actions happen at the same moment, they might all be successful, or one might fail. In this situation, differently annotated variants of the combination of the three actions lead to several possible new world states.

### The function $\tau$

As mentioned in the previous subsection, the function  $\tau$  computes the set of next world states which could result from performing a set of sub-actions and events during a time unit. One of these new world states is non-deterministically chosen by the global semantics. As sub-actions and events can interfere with each other, or lead to synergetic effects, and there are numerous combinations of sub-actions and events, specifying  $\tau$  in a concrete case is a vast amount of work. The definition of  $\tau$  depends on the physical actions and events possible in the domain of the multi-agent system, and  $\tau$  is thus one of the parameters of the skeleton programming language. We won't go into the technical details of how to concisely specify  $\tau$ . Other researchers have tried to give logical formalisations of the effects of actions in a dynamic environment. Issues like the frame, qualification and ramification problems play a role here, as well as theories about concurrent actions. We refer to Sandewall's and Shoham's work on non-monotonic logics for a detailed account of these issues [110], and to [7], for a promising alternative formalisation.

What we will do here is explain how  $\tau$  contributes to the objectives of

our operational model. The skeleton programming language and its semantics are aimed to provide a reasonably realistic model of interaction in multi-agent systems. We have individual actions and group actions, which have duration, can fail anytime during the execution and can interfere with each other or with events taking place. We start with the simpler case, which is individual action.

As we explained in Section 4.2.3 and Section 4.4.1, each individual action  $a \in \mathcal{A}_s$  is replaced by one of its extensions when it is semantically interpreted. For each possible extension, local traces are constructed, which are then combined at the global semantic level into a system trace. We will explain the role of  $\tau$  by using an example we introduced in Section 4.2.3. We have an action  $c$  with a duration of three time units (if successful). Imagine this action  $c$  is the action of pouring a cup of coffee. There are four possible extensions of this action, namely  $c_{1+}; c_{2+}; c_{3+}$ ,  $c_{1+}; c_{2+}; c_{3-}$ ,  $c_{1+}; c_{2-}$  and  $c_{1-}$ . If action  $c$  is part of the program of agent  $i$ , then the local semantics yields local traces with each of these four extensions. Thus, there are local traces where the action is performed successfully, and there are traces in which pouring a cup of coffee fails at the first, second or third time unit of the execution. Whether the action will succeed (first extension) or fail (other three extensions) is determined in the global semantics, and  $\tau$  plays a major role in this. We look at one specific moment in time, at which the execution of  $c$  starts. The local traces of the agent under focus thus start with an external transition step  $S \xrightarrow{i, (O, C, P)} S'$ , where either  $c_{1+} \in P$  (in case one of the first three extensions is executed) or  $c_{1-} \in P$  (in case the fourth extension is picked). During this time unit, agent  $i$  might be doing other actions, so  $P$  can contain more sub-actions, and the other agents might also be acting. Also, events could be happening. The success or failure of  $c_1$  depends on all these other occurrences, and on the state of the world; the function  $\tau$  formalises this. It takes the world state, the set of all atomic sub-actions and the events taking place during the time unit. Suppose the world circumstances are ideal for the first sub-action (the cup and the coffee pot are in the right position, the cup is not (completely) full yet and there is coffee in the pot) and there are no interfering occurrences. Thus,  $c_1$  will certainly succeed. This means that  $\tau$  will yield the empty set of successor world states for each action set containing  $c_{1-}$ . The global transition rule doesn't yield a global transition if the extension  $c_{1-}$  is part of the program to be executed of agent  $i$ . Only the other three extensions thus contribute to a system trace; because the first step has been taken, the remainders left to be executed are  $c_{2+}; c_{3+}$ ,  $c_{2+}; c_{3-}$  and  $c_{2-}$ , respectively. Now suppose during the second time unit of the execution, another agent gently touches the arm-actuator of agent  $i$ , because it wants to attract  $i$ 's attention. These circumstances both allow success or failure

of the second sub-action  $c_2$ , as it is not clear whether the pat on the actuator will affect agent  $i$ 's coffee-pouring. This means that all three remaining sequences can contribute to the global trace. The result of  $\tau$  will be different, depending on whether  $c_{2+}$  or  $c_{2-}$  is part of the action set (successful actions lead to effects different from the effects of failing actions). In the first case, the cup will continue to be filled with coffee, while in the second case, coffee is spilled all over the environment of the coffee cup. In the global traces where the second sub-action fails, execution of  $c$  is over; the third sub-action isn't executed. In case the second action succeeded, there are again two possibilities for the third sub-action. Suppose the circumstances cause the third sub-action to fail, for example because the cup initially contained some coffee already and thus will overflow now. Only the action  $c_{3-}$  will contribute to the global trace. The overall result after executing the coffee pouring action is that only two of the four extensions become part of global traces, namely  $c_{1+}; c_{2+}; c_{3-}$  and  $c_{1+}; c_{2-}$ .

If two or more individual actions take place during overlapping time intervals, this might cause failure. Then, only failing extensions of the actions become part of the global traces, and the world state is transformed differently than would be the case if the actions succeeded. The actions can also lead to synergetic effects. Then, the actions don't fail, but yield effects other than the union of effects of the individual actions. For example, if two soccer-playing robots kick against the ball in the same direction, the ball moves faster across the playing field than when only one agent kicks it.

Group actions for the most part are treated identically to individual actions. A group action is replaced by one of its extensions. There can be extensions originating from different group action schemes, and they can fail at each time unit of the execution, just like the extensions of individual actions. Whether a certain sub-action of a contribution to a group action will succeed or fail is determined by the world situation, and the other occurrences taking place, analogously to the way sub-actions of individual actions succeed or fail. The only somewhat special feature of group actions is that if there is a group of agents executing group action contributions coming from one group action scheme, such that each contribution from the scheme is done by one agent, if they do this synchronised according to the group action scheme, if there are no other disturbing occurrences in the world and if the world situation is suitable for execution of the group action, then there will probably be *synergetic effects*. This is also formalised in the definition of  $\tau$ . Agents perform group actions mostly to achieve world situations which they can't realise on their own, so a well-coordinated and successful group action execution usually has some synergetic effects. For the definition of  $\tau$ , this merely means that the right synergetic effects have to be established in case the action set of the argument contains sub-actions that

are part of contributions to a group action coming from a certain group action scheme.

To illustrate this, we again use an example from Subsection 4.2.3. The group action we looked at there is LiftTable. We had the following group action scheme for this action:

[LiftLeg0-15;	LiftLeg15-30;	LiftLeg30-45;	skip;	skip,
skip;	skip;	LiftLeg0-15;	LiftLeg15-30;	LiftLeg30-45,
skip;	LiftLeg0-15;	LiftLeg15-30;	LiftLeg30-45;	skip,
skip;	skip;	LiftLeg0-15;	LiftLeg15-30;	LiftLeg30-45]

Now, suppose there is a bowl of soup on the table. If there are four agents executing contributions from the group action scheme in a synchronised manner, then no soup will spill out of the bowl, unless there are other circumstances in the world causing the soup to spill. We exclude this last circumstance for the sake of clarity. For the function  $\tau$ , this means that the combined execution of the first four sub-actions LiftLeg0-15, skip, skip and skip (first column of the group action scheme) should be successful (so only LiftLeg0-15+ is executable) and also lead to the soup staying in the bowl in the resulting world situation. The same holds for the next time instants.

In case the group members execute contributions from one group action scheme that are not properly synchronised (one agent starts two time units earlier with its contribution than the other agents), the group action probably won't yield the desired effects. But the individual contributions of the agents could still be successful. So, if one agent starts lifting its table-leg too early, then the soup will spill and the bowl will fall off the table, but the table-leg that agent is holding is lifted, as is the expected outcome of the individual contribution of that agent. Thus, the only special feature of group actions is that synergy can be obtained if the agents are properly synchronised.

## 4.5 Proof of correct reason–interact behaviour

When we defined the syntax of agent programs in Section 4.3.3, we were careful not to compose internal, reasoning statements (*Intractions*) and external, interacting statements (*Interactions*) in an arbitrary manner. We defined sets  $(\mathcal{B}_i, i \in \mathcal{I})$  of behaviours, which we claimed to be program fragments that, when executed, always result in all internal statements preceding all external statements. In an agent program, a behaviour is iterated, to form the reason–interact cycle of the agent. In this subsection, we will prove that indeed each execution of a behaviour leads to reasoning preceding acting.

In order to do this, we first have to formalise the concept of execution of

a program, or rather, of a composite statement. We also define the set of all composite statements.

**DEFINITION 4.19 (Composite statements)**

The set  $Comp\mathcal{S}_i$  of all composite statements related to agent  $i$  is the smallest set containing:

- \* skip
- \*  $a$ , where  $a \in \mathcal{A}_s \cup \mathcal{A}_g \cup \mathcal{A}_o \cup \mathcal{A}_{c(i)} \cup \mathcal{A}_{m(i)} \cup \mathcal{A}_{ex}$
- \*  $\varphi?$ , where  $\varphi \in \mathcal{L}_{rep(i)}$
- \*  $\alpha; \beta, \alpha + \beta, \alpha \parallel \beta, \alpha^+$ , where  $\alpha, \beta \in Comp\mathcal{S}_i$

Composite statements are simply compositions of basic statements of the skeleton programming language, without any syntactic restriction. We have  $\mathcal{S}_i \subset \mathcal{S}_i^- \subset Comp\mathcal{S}_i$ .

Next, we define *traces*. A trace of a composite statement is any sequence of transitions that can result when executing the statement, either partially or completely. As statements can be part of a larger composite statement (a program or program remainder), we need a recursive definition for this. We also formally define *transitions*.

**DEFINITION 4.20 (Transitions and traces)**

If  $\langle \sigma, \delta, \pi \rangle$  and  $\langle \sigma', \delta', \pi' \rangle$  are local agent configurations of agent  $i$  and  $\xrightarrow{l}$  is either an internal transition arrow labelled with the agent identity  $i$  or an external transition arrow labelled with the agent identity  $i$  and a triple of interaction sets  $(O, C, P)$ , and  $\langle \sigma, \delta, \pi \rangle \xrightarrow{l} \langle \sigma', \delta', \pi' \rangle$ , is deducible by means of the transition system in Section 4.4.2, then  $\langle \sigma, \delta, \pi \rangle \xrightarrow{l} \langle \sigma', \delta', \pi' \rangle$  is a *transition*.

In the following, we will use the *convention* to equate composite statements  $\pi$  to  $\pi_0$ . The use of this convention will become apparent soon. A *trace* of a composite statement  $\alpha \in Comp\mathcal{S}_i$  is recursively defined as follows, equating  $\alpha$  to  $\alpha_0$ :

- \*  $\langle \sigma_0, \delta_0, \alpha_0 \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \alpha_1 \rangle \xrightarrow{l_2} \langle \sigma_2, \delta_2, \alpha_2 \rangle \xrightarrow{l_3} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \alpha_m \rangle$ , where each  $\langle \sigma_{k-1}, \delta_{k-1}, \alpha_{k-1} \rangle \xrightarrow{l_k} \langle \sigma_k, \delta_k, \alpha_k \rangle$  is a transition. In case  $\alpha_m = \surd$ , we call the trace *successful*; otherwise we call the trace *partial*.

\* if  $\langle \sigma_0, \delta_0, \pi_0 \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \pi_1 \rangle \xrightarrow{l_2} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \pi_m \rangle$  is a (successful or partial) trace of  $\alpha$ , then so are the following:

- 1  $\langle \sigma_0, \delta_0, \pi_0; \beta \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \pi_1; \beta \rangle \xrightarrow{l_2} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \pi_m; \beta \rangle$
- 2  $\langle \sigma_0, \delta_0, \beta + \pi_0 \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \pi_1 \rangle \xrightarrow{l_2} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \pi_m \rangle$
- 3  $\langle \sigma_0, \delta_0, \pi_0 + \beta \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \pi_1 \rangle \xrightarrow{l_2} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \pi_m \rangle$
- 4  $\langle \sigma_0, \delta_0, \pi_0^+ \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \pi_1 \rangle \xrightarrow{l_2} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \pi_m \rangle$
- 5  $\langle \sigma_0, \delta_0, \pi_0^+ \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \pi_1; \pi_0^+ \rangle \xrightarrow{l_2} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \pi_m; \pi_0^+ \rangle$

where  $\beta \in \text{Comp}\mathcal{S}_i$  is another composite statement.

Thus, a trace of a composite statement  $\alpha$  is a number of consecutive transitions that result when executing  $\alpha$ . A trace of  $\alpha$  is successful if  $\alpha$  is wholly executed, and partial if there is a remainder of  $\alpha$  left to be executed. At first sight, the definition of trace-hood might seem hard to grasp, because of its recursive nature. Intuitively, the first \* might seem to capture the idea of trace-hood completely, making the second \* superfluous. But this second \* does make sense. The second \* in the definition above postulates that when  $\alpha$  is part of a larger composite statement, in such a way that  $\alpha$  is executed before the rest of the composite statement, then the resulting sequence of transitions is also a trace of  $\alpha$ . We need this in our proofs later on. As an example of this, look at the statement  $(\mathbf{a}; \beta)^+$ , where  $\mathbf{a} \in \mathcal{A}_s$ ,  $\mathbf{a} \triangleleft \mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}$ , and  $\beta \in \text{Comp}\mathcal{S}_i$  is some composite statement. Before giving semantics to  $\mathbf{a}$ , it is replaced by its extension. By convention, a trace of an extension of a physical action is regarded as a trace of the action itself. Now,  $\langle \sigma_0, \delta_0, \mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+} \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{1+}\})}$

$\langle \sigma_1, \delta_0, \mathbf{a}_{2+}; \mathbf{a}_{3+} \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{2+}\})} \langle \sigma_2, \delta_0, \mathbf{a}_{3+} \rangle$  is a trace of  $\mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}$  according to the first \* of Definition 4.20, and thus a trace of  $\mathbf{a}$ . The second \* then allows to derive that

$$\langle \sigma_0, \delta_0, \mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{1+}\})} \langle \sigma_1, \delta_0, \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{2+}\})} \langle \sigma_2, \delta_0, \mathbf{a}_{3+}; \beta \rangle$$

and

$$\langle \sigma_0, \delta_0, (\mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta)^+ \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{1+}\})} \langle \sigma_1, \delta_0, \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta; (\mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta)^+ \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{2+}\})} \langle \sigma_2, \delta_0, \mathbf{a}_{3+}; \beta; (\mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta)^+ \rangle$$

are also traces of  $\mathbf{a}$ . Without the second \*, the last trace would have been a trace of  $(\mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta)^+$ , but not of  $\mathbf{a}$ .

Note that trace-hood doesn't survive parallel composition; if  $\alpha$  and  $\beta$  are executed in parallel, then the resulting trace doesn't start with a trace of  $\alpha$  or  $\beta$  exclusively, as the steps of  $\alpha$  and  $\beta$  either are interleaved or accumulated. This is why Definition 4.20 doesn't include a case for the  $\parallel$ -operator.

Now, we can phrase the central theorem of this subsection:

**THEOREM 4.1** (Behaviours produce internal steps followed by external steps)  
 $\forall i \in \mathcal{I} \forall \gamma \in \mathcal{B}_i$  : each trace of  $\gamma$  starts with zero or more internal transitions, followed by zero or more external transitions.

This theorem describes a property of the transition arrows of the traces of behaviours. The sense buffers and the mental states in agent configurations are not relevant for properties like this. The agent program, the last element of agent configurations, is relevant, as it is the agent program that determines to a large extent the set of traces resulting when executing the program. The only time that the sense buffer or the mental state can influence the trace of a composite statement is when the program contains mental tests ( $\varphi?$ ) or mental update actions (from  $\mathcal{A}_{m(i)}$ ). Tests don't influence the nature (external or internal) of transitions; they can only lead to termination of a trace (failure:  $\varphi? \not\rightarrow$ ) because the condition isn't true of the sense buffer and the mental state. Mental updates also don't influence the nature of the transitions; they merely lead to a new mental state.

Because of this, we define the notion of *abstract traces*, in which sense buffers and mental states are left out. In the proof of Theorem 4.1, we will make use of these abstract traces, which simplifies matters.

**DEFINITION 4.21** (Abstract transitions and traces)

If there are  $\sigma, \sigma' \subseteq \mathcal{L}_{sb(i)}$  and  $\delta, \delta' \subseteq \mathcal{L}_{m(i)}$  such that  $\langle \sigma, \delta, \pi \rangle \xrightarrow{l} \langle \sigma', \delta', \pi' \rangle$  is an transition, then  $\pi \xrightarrow{l} \pi'$  is an *abstract transition*.

An *abstract trace* of composite statement  $\alpha \in \text{Comp}\mathcal{S}_i$  is recursively defined as follows, equating  $\alpha$  to  $\alpha_0$ :

- \*  $\alpha_0 \xrightarrow{l_1} \alpha_1 \xrightarrow{l_2} \alpha_2 \xrightarrow{l_3} \dots \xrightarrow{l_m} \alpha_m$ , where each  $\alpha_{k-1} \xrightarrow{l_k} \alpha_k$  is an abstract transition. In case  $\alpha_m = \surd$ , we call the abstract trace successful, and otherwise we call it partial.
- \* if  $\pi_0 \xrightarrow{l_1} \pi_1 \xrightarrow{l_2} \dots \xrightarrow{l_m} \pi_m$  is a (successful or partial) trace of  $\alpha$ , then so are the following:
  - 1  $\pi_0; \beta \xrightarrow{l_1} \pi_1; \beta \xrightarrow{l_2} \dots \xrightarrow{l_m} \pi_m; \beta$

- 2  $\beta + \pi_0 \xrightarrow{l_1} \pi_1 \xrightarrow{l_2} \dots \xrightarrow{l_m} \pi_m$
- 3  $\pi_0 + \beta \xrightarrow{l_1} \pi_1 \xrightarrow{l_2} \dots \xrightarrow{l_m} \pi_m$
- 4  $\pi_0^+ \xrightarrow{l_1} \pi_1 \xrightarrow{l_2} \dots \xrightarrow{l_m} \pi_m$
- 5  $\pi_0^+ \xrightarrow{l_1} \pi_1; \pi_0^+ \xrightarrow{l_2} \dots \xrightarrow{l_m} \pi_m; \pi_0^+$

where  $\beta \in \text{Comp}\mathcal{S}_i$  is another composite statement.

For each ordinary, full trace, there is exactly one matching abstract trace, which is obtained by leaving out the sense buffers and mental states. For example, the abstract version of

$$\langle \sigma_0, \delta_0, \mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{1+}\})} \langle \sigma_1, \delta_0, \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta \rangle \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{2+}\})} \langle \sigma_2, \delta_0, \mathbf{a}_{3+}; \beta \rangle$$

is the abstract trace

$$\mathbf{a}_{1+}; \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{1+}\})} \mathbf{a}_{2+}; \mathbf{a}_{3+}; \beta \xrightarrow{i, (\emptyset, \emptyset, \{\mathbf{a}_{2+}\})} \mathbf{a}_{3+}; \beta.$$

Note that the traces we used in Example 4.4, Section 4.4.2 were also abstract. Though there is exactly one abstract trace for every full trace, this is not true the other way around. For example,  $\text{del}(\mathbf{B}_i(\varphi)); \mathbf{B}_i(\varphi)? \xrightarrow{i} \mathbf{B}_i(\varphi)? \xrightarrow{i} \surd$  is an abstract trace, for which there are no corresponding full traces. This is immediately clear when we look at one candidate full trace (we assume the revision function for the mental state works as one would expect here):

$$\langle \sigma, \{\mathbf{B}_i(\varphi)\}, \text{del}(\mathbf{B}_i(\varphi)); \mathbf{B}_i(\varphi)? \rangle \xrightarrow{i} \langle \sigma', \emptyset, \mathbf{B}_i(\varphi)? \rangle \not\xrightarrow{i}.$$

Also, some abstract traces have many corresponding full traces. An example of such an abstract trace is

$$\text{ins}(\mathbf{B}_i(\varphi)); \mathbf{B}_i(\varphi)? \xrightarrow{i} \mathbf{B}_i(\varphi)? \xrightarrow{i} \surd.$$

We now have the following lemma:

LEMMA 4.1 (Abstract traces suffice)

In order to prove Theorem 4.1, it is enough to prove the property mentioned there for abstract traces instead of for full traces.

PROOF (sketchy):

Take a full trace of a behaviour  $\gamma \in \mathcal{B}_i$ . Take the corresponding abstract trace, by leaving out the sense buffers and mental states from the agent configurations. The nature of the transitions (whether the transition is internal or external) hasn't changed by this abstraction step. So, if we prove that the abstract trace consists of zero or more internal transitions followed by zero or more external transitions, then we have also proven this for the full trace.

End of PROOF.

Because of this lemma, all subsequent theorems, which contribute to establishing the proof of Theorem 4.1, will pertain to abstract traces instead of to full ones. Usually we will be sloppy and write ‘traces’ instead of ‘abstract traces’ and ‘transitions’ instead of ‘abstract transitions’.

We can simplify matters even further, as the second lemma shows:

**LEMMA 4.2 (Irrelevance of program context)**

In order to prove a property of the transition arrows in all (full or partial) traces of a statement  $\alpha$ , it suffices to prove this property for the subset of traces of  $\alpha$  defined at the first \* of Definitions 4.20 and 4.21. The property then also holds for all traces of  $\alpha$  in which  $\alpha$  is part of a larger composite statement (defined at the second \* of Definitions 4.20 and 4.21).

**PROOF (sketchy):**

This lemma immediately follows from Definitions 4.20 and 4.21, because the recursive part of these definitions (second \*) copies the transition arrows  $\xrightarrow{l_1}, \xrightarrow{l_2}, \dots, \xrightarrow{l_m}$  of the trace starting at statement  $\pi_0$  to traces starting at composite statements where  $\pi_0$  is one of the constituent statements.

End of PROOF

This lemma releases us from having to use induction on the structure of the program component in the starting configuration of traces, in order to prove a property of transition arrows in traces.

Because of the way behaviours are defined (Definition 4.11), we need theorems regarding traces of statements from *Interactions<sub>i</sub>* and *Intractions<sub>i</sub>*. We will prove these theorems later on.

**THEOREM 4.2 (Interactions produce external steps)**

$\forall i \in \mathcal{I} \forall \alpha \in \text{Interactions}_i$  : each trace of  $\alpha$  consists of zero or more external transitions.

**THEOREM 4.3 (Intractions produce internal steps)**

$\forall i \in \mathcal{I} \forall \alpha \in \text{Intractions}_i$  : each trace of  $\alpha$  consists of zero or more internal transitions.

In order to prove the theorems above, we give some more theorems, regarding traces of statements composed using the operators ‘;’, ‘+’ and ‘+’. In the sequel, we will sometime equate statement  $\pi$  to  $\pi_0$ , as we did before.

**THEOREM 4.4 (Traces of sequential compositions)**

$\forall i \in \mathcal{I} \forall \alpha, \beta \in \text{Comp}\mathcal{S}_i$  : each trace of  $\alpha; \beta$  either is a trace of  $\alpha$  or a successful trace of  $\alpha$ , followed by a trace of  $\beta$ .

**PROOF:**

According to Lemma 4.2, we only have to look at the subset of traces of  $\alpha; \beta$  that start with  $\alpha; \beta \xrightarrow{l}$ . Taking into account the transition rule for sequential composition, such a trace can have two forms, as execution of  $\beta$  can either have started after successful execution of  $\alpha$  (second case) or  $\beta$  is still to be executed, after partial or successful execution of  $\alpha$  (first case).

- \*  $\alpha_0; \beta \xrightarrow{l_1} \alpha_1; \beta \xrightarrow{l_2} \dots \xrightarrow{l_k} \alpha_k; \beta$ , where  $\alpha_{j-1} \xrightarrow{l_j} \alpha_j$ ,  $1 \leq j \leq k$  are transitions. Then,  $\alpha_0 \xrightarrow{l_1} \alpha_1 \xrightarrow{l_2} \dots \xrightarrow{l_k} \alpha_k$  is a trace of  $\alpha$  and thus, according to •1 of Definition 4.21, so is  $\alpha_0; \beta \xrightarrow{l_1} \alpha_1; \beta \xrightarrow{l_2} \dots \xrightarrow{l_k} \alpha_k; \beta$ .
- \*  $\alpha_0; \beta \xrightarrow{l_1} \alpha_1; \beta \xrightarrow{l_2} \dots \xrightarrow{l_m} \surd; \beta = \beta_0 \xrightarrow{l_{m+1}} \beta_1 \xrightarrow{l_{m+2}} \dots \xrightarrow{l_{m+n}} \beta_n$ , where  $\alpha_{j-1} \xrightarrow{l_j} \alpha_j$ ,  $1 \leq j \leq m$ ,  $\alpha_m = \surd$  and  $\beta_{j-1} \xrightarrow{l_{m+j}} \beta_j$ ,  $1 \leq j \leq n$  are transitions. Then,  $\alpha_0 \xrightarrow{l_1} \alpha_1 \xrightarrow{l_2} \dots \xrightarrow{l_m} \surd$  is a successful trace of  $\alpha$  and thus, according to •1 of Definition 4.21, so is  $\alpha_0; \beta \xrightarrow{l_1} \alpha_1; \beta \xrightarrow{l_2} \dots \xrightarrow{l_m} \surd; \beta$ . The sequence of transitions  $\beta_0 \xrightarrow{l_{m+1}} \beta_1 \xrightarrow{l_{m+2}} \dots \xrightarrow{l_{m+n}} \beta_n$  is a trace of  $\beta$ , so  $\alpha_0; \beta \xrightarrow{l_1} \alpha_1; \beta \xrightarrow{l_2} \dots \xrightarrow{l_m} \surd; \beta = \beta_0 \xrightarrow{l_{m+1}} \beta_1 \xrightarrow{l_{m+2}} \dots \xrightarrow{l_{m+n}} \beta_n$  is a successful trace of  $\alpha$ , followed by a trace of  $\beta$ .

End of PROOF

**THEOREM 4.5 (Traces of compositions by choice)**

$\forall i \in \mathcal{I} \forall \alpha, \beta \in \text{Comp}\mathcal{S}_i$  : each trace of  $\alpha + \beta$  either is a trace of  $\alpha$  or a trace of  $\beta$ .

**PROOF:**

According to Lemma 4.2, we only have to look at the subset of traces of  $\alpha + \beta$  that start with  $\alpha + \beta \xrightarrow{l}$ . Taking into account the transition rules for non-deterministic choice, such a trace can have two forms:

- \*  $\alpha_0 + \beta_0 \xrightarrow{l_1} \alpha_1 \xrightarrow{l_2} \alpha_2 \xrightarrow{l_3} \dots \xrightarrow{l_m} \alpha_m$ , where  $\alpha_0 \xrightarrow{l_1} \alpha_1$  is a transition. Then,  $\alpha_0 \xrightarrow{l_1} \alpha_1 \xrightarrow{l_2} \alpha_2 \xrightarrow{l_3} \dots \xrightarrow{l_m} \alpha_m$  is a trace of  $\alpha$  and thus, according to  $\bullet 3$  of Definition 4.21, so is  $\alpha_0 + \beta_0 \xrightarrow{l_1} \alpha_1 \xrightarrow{l_2} \alpha_2 \xrightarrow{l_3} \dots \xrightarrow{l_m} \alpha_m$ .
- \*  $\alpha_0 + \beta_0 \xrightarrow{l'_1} \beta_1 \xrightarrow{l'_2} \beta_2 \xrightarrow{l'_3} \dots \xrightarrow{l'_n} \beta_n$ , where  $\beta_0 \xrightarrow{l'_1} \beta_1$  is a transition. Then,  $\beta_0 \xrightarrow{l'_1} \beta_1 \xrightarrow{l'_2} \beta_2 \xrightarrow{l'_3} \dots \xrightarrow{l'_n} \beta_n$  is a trace of  $\beta$  and thus, according to  $\bullet 2$  of Definition 4.21, so is  $\alpha_0 + \beta_0 \xrightarrow{l'_1} \beta_1 \xrightarrow{l'_2} \beta_2 \xrightarrow{l'_3} \dots \xrightarrow{l'_n} \beta_n$ .

As these are the only two possibilities, every trace of  $\alpha + \beta$  is either a trace of  $\alpha$  or a trace of  $\beta$ .

End of PROOF

#### THEOREM 4.6 (Traces of iterated compositions)

$\forall i \in \mathcal{I} \forall \alpha \in \text{CompS}_i$  : each trace of  $\alpha^+$  is a concatenation of  $n+1$ ,  $n \geq 0$  traces of  $\alpha$ , the first  $n$  of which are successful.

PROOF:

According to Lemma 4.2, we only have to look at the subset of traces of  $\alpha^+$  that start with  $\alpha^+ \xrightarrow{l}$ . There are two transition rules for iteration, and at the start of every round of the iteration one of these rules is chosen. In case the second rule is chosen, then there always is a potential next round. This next round is started if execution of  $\alpha$  in the current round terminates successfully. We show the construction of a trace of  $\alpha^+$ , where the second transition rule is used the first  $n$  rounds. In the final round, either the first or the second transition rule is applied. This construction is the most general way to arrive at a trace of  $\alpha^+$ .

We start by looking at the first of the  $n$  rounds. We have the trace  $\alpha^+ \xrightarrow{l'_1} \alpha^+_1; \alpha^+ \xrightarrow{l'_2} \alpha^+_2; \alpha^+ \xrightarrow{l'_3} \dots \xrightarrow{l'_{m_1}} \sqrt{}; \alpha^+ = \alpha^+$ , where  $\alpha_0 \xrightarrow{l'_1} \alpha^+_1$  and  $\alpha^+_{j-1} \xrightarrow{l'_j} \alpha^+_j$ ,  $2 \leq j \leq m_1$ ,  $\alpha^+_{m_1} = \sqrt{} are transitions. In constructing this trace, we first use the second rule for iteration, and then  $m_1 - 1$  times the rule for sequential composition. We then have that  $\alpha \xrightarrow{l'_1} \alpha^+_1 \xrightarrow{l'_2} \alpha^+_2 \xrightarrow{l'_3} \dots \xrightarrow{l'_{m_1}} \sqrt{} is a successful trace of  $\alpha$ . It is not possible to take an unsuccessful, partial trace of  $\alpha$  here, as then the first round of the execution of  $\alpha^+$  wouldn't result in  $\alpha^+$ , but in  $\alpha_r; \alpha^+$ , and thus the next round can't be started. Because of  $\bullet 5$  of Definition 4.21,$$

$\alpha^+ \xrightarrow{l_1^1} \alpha_1^+; \alpha^+ \xrightarrow{l_2^1} \alpha_2^+; \alpha^+ \xrightarrow{l_3^1} \dots \xrightarrow{l_{m_1}^1} \sqrt{}; \alpha^+ = \alpha^+$  is a successful trace of  $\alpha$ .

After this first round, the statement to be executed again is  $\alpha^+$ , and the next round starts, again producing a successful trace of  $\alpha$  (which might be different than the one in the first round). After  $n$  rounds, the trace of  $\alpha^+$  consists of a concatenation of  $n$  traces of  $\alpha$ :  $\alpha^+ \xrightarrow{l_1^1} \dots \xrightarrow{l_{m_1}^1} \alpha^+ \xrightarrow{l_1^2} \dots \xrightarrow{l_{m_2}^2} \alpha^+ \xrightarrow{l_1^3} \dots \xrightarrow{l_{m_3}^3} \alpha^+ \dots \xrightarrow{l_1^n} \dots \xrightarrow{l_{m_n}^n} \alpha^+$ . Then, round  $n + 1$  starts. There are two options now, depending on whether the first or second transition rule for iteration is used:

- \* In case the first rule is used, a trace of the last round is  $\alpha^+ \xrightarrow{l_1^{n+1}} \alpha_1^{n+1} \xrightarrow{l_2^{n+1}} \dots \xrightarrow{l_k^{n+1}} \alpha_k^{n+1}$  for some  $k$ , where  $\alpha_0 \xrightarrow{l_1^{n+1}} \alpha_1^{n+1}$  is a transition. Then,  $\alpha_0 \xrightarrow{l_1^{n+1}} \alpha_1^{n+1} \xrightarrow{l_2^{n+1}} \dots \xrightarrow{l_k^{n+1}} \alpha_k^{n+1}$  is a trace of  $\alpha$ , and according to **•4** of Definition 4.21, then so is  $\alpha^+ \xrightarrow{l_1^{n+1}} \alpha_1^{n+1} \xrightarrow{l_2^{n+1}} \dots \xrightarrow{l_k^{n+1}} \alpha_k^{n+1}$ .
- \* In case the second rule is used, a trace of the last round is  $\alpha^+ \xrightarrow{l_1^{n+1}} \alpha_1^{n+1}; \alpha^+ \xrightarrow{l_2^{n+1}} \dots \xrightarrow{l_k^{n+1}} \alpha_k^{n+1}; \alpha^+$  for some  $k$ , where  $\alpha_0 \xrightarrow{l_1^{n+1}} \alpha_1^{n+1}$  and  $\alpha_{j-1}^{n+1} \xrightarrow{l_j^{n+1}} \alpha_j^{n+1}, 2 \leq j \leq k$  are transitions (these transitions might be different from those in the previous case, even though we used the same indices for labels and program remainders). Then,  $\alpha_0 \xrightarrow{l_1^{n+1}} \alpha_1^{n+1} \xrightarrow{l_2^{n+1}} \dots \xrightarrow{l_k^{n+1}} \alpha_k^{n+1}$  is a trace of  $\alpha$ , and according to **•5** of Definition 4.21, then so is  $\alpha^+ \xrightarrow{l_1^{n+1}} \alpha_1^{n+1}; \alpha^+ \xrightarrow{l_2^{n+1}} \dots \xrightarrow{l_k^{n+1}} \alpha_k^{n+1}; \alpha^+$

This most general construction of a trace of  $\alpha^+$  shows that all such traces are concatenations of  $n + 1, n \geq 0$  traces of  $\alpha$ , of which the first  $n$  traces are successful.

End of PROOF

Using Theorems 4.4, 4.5 and 4.6, we can prove Theorems 4.2 and 4.3.

**PROOF OF THEOREM 4.2 (Interactions produce external steps)**

Take a statement  $\pi \in \text{Interactions}_i, i \in \mathcal{I}$ . Because of Lemma 4.2, in order to prove that all traces of  $\pi$  are purely external, we only have to look at traces starting with  $\pi \xrightarrow{l}$ . The proof is by induction on the structure of  $\pi$ .

**Induction basis:** There are several cases:

$\pi = \text{skip}$ : For the statement **skip**, only one trace is possible, namely  $\text{skip} \xrightarrow{i,(\emptyset,\emptyset,\emptyset)} \checkmark$ , which clearly consists of only external transitions.

$\pi = \mathbf{a} \in \mathcal{A}_s \cup \mathcal{A}_g$ : Physical actions extend into sequences of failing or succeeding sub-actions. Suppose  $\mathbf{a} \triangleleft \mathbf{a}_1; \mathbf{a}_2; \dots; \mathbf{a}_p$ . Defining  $(S \xrightarrow{l} S)^*$  to mean zero or more applications of  $S \xrightarrow{l} S$ , we have the following successful trace of  $\mathbf{a}$ :

$$(\mathbf{a}_1; \mathbf{a}_2; \dots; \mathbf{a}_p \xrightarrow{i,(\emptyset,\emptyset,\emptyset)} \mathbf{a}_1; \mathbf{a}_2; \dots; \mathbf{a}_p)^* \xrightarrow{i,(\emptyset,\emptyset,\{\mathbf{a}_1\})} \mathbf{a}_2; \dots; \mathbf{a}_p \xrightarrow{i,(\emptyset,\emptyset,\{\mathbf{a}_2\})} \dots \xrightarrow{i,(\emptyset,\emptyset,\{\mathbf{a}_{p-1}\})} \mathbf{a}_p \xrightarrow{\emptyset,\emptyset,\{\mathbf{a}_p\}} \checkmark.$$

Partial traces are prefixes of these traces. So, each trace of  $\mathbf{a}$  (which is a trace of an extension of  $\mathbf{a}$ ) consists of zero or more external transitions.

$\pi = \mathbf{o} \in \mathcal{A}_o$ : For observative actions, successful traces are:

$$(\mathbf{o} \xrightarrow{i,(\emptyset,\emptyset,\emptyset)} \mathbf{o})^* \xrightarrow{i,(\{\mathbf{O}\},\emptyset,\emptyset)} \checkmark. \text{ Partial traces are prefixes of these, so all traces of } \mathbf{o} \text{ are purely external.}$$

$\pi = \mathbf{c} \in \mathcal{A}_{c(i)}$ : For communicative actions, successful traces are:

$$(\mathbf{c} \xrightarrow{i,(\emptyset,\emptyset,\emptyset)} \mathbf{c})^* \xrightarrow{i,(\emptyset,\{\mathbf{C}\},\emptyset)} \checkmark. \text{ Partial traces are prefixes of these, so all traces of } \mathbf{c} \text{ are purely external.}$$

**Induction Hypothesis:** For simpler statements, all traces consist of zero or more external transitions.

**Induction step:** As there are three ways to compose interactions, we have three cases:

$\pi = \alpha; \beta$ : Theorem 4.4 gives us that each trace of  $\alpha; \beta$  either is a trace of  $\alpha$  or a successful trace of  $\alpha$  followed by a trace of  $\beta$ . Because of the induction hypothesis, all traces of  $\alpha$  and  $\beta$  are purely external. All traces of  $\alpha; \beta$  thus are also purely external and consists of zero or more external transitions.

$\pi = \alpha + \beta$ : Theorem 4.5 gives us that each trace of  $\alpha + \beta$  either is a trace of  $\alpha$  or a trace of  $\beta$ . Because of the induction hypothesis,

all traces of  $\alpha$  and  $\beta$  are purely external. All traces of  $\alpha + \beta$  thus are also purely external and consists of zero or more external transitions.

$\pi = \alpha \parallel \beta$ : All traces of  $\alpha$  and  $\beta$  only contain external transitions. This means that when computing a trace of  $\alpha \parallel \beta$ , only the first transition rule for parallel composition is applicable. As this rule always results in external transitions, all traces of  $\alpha \parallel \beta$  are purely external and consists of zero or more external transitions.

End of PROOF

### PROOF OF THEOREM 4.3 (Intractions produce internal steps)

Take a statement  $\pi \in \text{Intractions}_i, i \in \mathcal{I}$ . Because of Lemma 4.2, in order to prove that all traces of  $\pi$  are purely internal, we only have to look at traces starting with  $\pi \xrightarrow{l}$ . The proof is by induction on the structure of  $\pi$ .

**Induction basis:** There are two cases:

$\pi = \varphi?$ : There are two possible traces of  $\varphi?$ , namely  $\varphi? \xrightarrow{i} \surd$  in case  $\varphi$  holds, and  $\varphi? \not\rightarrow$ , in case  $\varphi$  doesn't hold (failure). The last trace consists of zero internal steps, and the first of one. Thus, all traces of  $\varphi?$  consist of zero or more internal transitions.

$\pi = m \in \mathcal{A}_{m(i)}$ : Mental updates have only one possible abstract trace, namely  $m \xrightarrow{i} \surd$ , which clearly is purely internal.

**Induction Hypothesis:** For simpler statements, all traces consist of zero or more internal transitions.

**Induction step:** As there are three ways to compose intractions, we have three cases:

$\pi = \alpha; \beta$ : Theorem 4.4 gives us that each trace of  $\alpha; \beta$  either is a trace of  $\alpha$  or a succesful trace of  $\alpha$  followed by a trace of  $\beta$ . Because of the induction hypothesis, all traces of  $\alpha$  and  $\beta$  are purely internal. All traces of  $\alpha; \beta$  thus are also purely internal and consist of zero or more internal transitions.

$\pi = \alpha + \beta$ : Theorem 4.5 gives us that each trace of  $\alpha + \beta$  either is a trace of  $\alpha$  or a trace of  $\beta$ . Because of the induction hypothesis, all traces of  $\alpha$  and  $\beta$  are purely internal. All traces of  $\alpha + \beta$  thus are also purely internal and consist of zero or more internal transitions.

$\pi = \alpha \parallel \beta$ : All traces of  $\alpha$  and  $\beta$  only contain internal transitions. This means that when computing a trace of  $\alpha \parallel \beta$ , only the second and third transition rule for parallel composition are applicable. As this rule always results in internal transitions, all traces of  $\alpha \parallel \beta$  are purely internal and consist of zero or more internal transitions.

End of PROOF

Now, we can prove the central theorem of this subsection:

PROOF OF THEOREM 4.1 (Correct reason–interact behaviours)

Take a behaviour  $\gamma \in \mathcal{B}_i, i \in \mathcal{I}$ . Because of Lemma 4.2, in order to prove that all traces of  $\gamma$  start with zero or more internal transitions, followed by zero or more external transitions, we only have to look at traces starting with  $\gamma \xrightarrow{l}$ . The proof is by induction on the structure of  $\gamma$ .

**Induction basis:** There are two cases:

$\gamma = \alpha, \alpha \in \text{Interactions}_i$ : According to Theorem 4.2 every trace of  $\alpha$  consists of zero or more external transitions. Thus, every trace of  $\alpha$  has zero internal transitions, followed by zero or more external transitions, which proves the theorem for this case.

$\gamma = \beta, \beta \in \text{Intractions}_i$ : According to Theorem 4.3 every trace of  $\beta$  consists of zero or more internal transitions. Thus, every trace of  $\beta$  has zero or more internal transitions, followed by zero external transitions, which proves the theorem for this case.

**Induction Hypothesis:** For simpler behaviours, all traces consist of a sequence of internal transitions, followed by a sequence of external transitions.

**Induction step:** As there are four ways to compose simpler behaviours into more complex behaviours, we discern four cases:

$\gamma = \alpha; \beta$ , where  $\alpha \in \text{Intractions}_i$  and  $\beta \in \mathcal{B}_i$ : Theorem 4.4 gives us that each trace of  $\alpha; \beta$  either is a trace of  $\alpha$  or a successful trace of  $\alpha$  followed by a trace of  $\beta$ . As  $\alpha \in \text{Intractions}_i$ , each trace of  $\alpha$  is purely internal (Theorem 4.3). So, in case a trace of  $\alpha; \beta$  is a trace of  $\alpha$ , it is purely internal, and thus satisfies the property that it has zero or more internal transitions, followed by zero or more external transitions. Because of the induction hypothesis, all traces of  $\beta$  consist of a number of internal transitions followed by a number of external transitions. Thus, in case a trace of  $\alpha; \beta$

is a successful trace of  $\alpha$  followed by a trace of  $\beta$ , it consists of the internal steps of  $\alpha$ , followed by first the internal steps of  $\beta$  and then the external steps of  $\beta$ . The resulting trace has a number of internal steps followed by a number of external steps.

$\gamma = \alpha; \beta$ , where  $\alpha \in \mathcal{B}_i$  and  $\beta \in \text{Interactions}_i$ : Theorem 4.4 gives us that each trace of  $\alpha; \beta$  either is a trace of  $\alpha$  or a successful trace of  $\alpha$  followed by a trace of  $\beta$ . Because of the induction hypothesis, all traces of  $\alpha$  consist of a number of internal transitions followed by a number of external transitions. So, in case a trace of  $\alpha; \beta$  is a trace of  $\alpha$ , it satisfies the property that it has zero or more internal transitions, followed by zero or more external transitions. As  $\beta \in \text{Interactions}_i$ , each trace of  $\beta$  is purely external (Theorem 4.2). Thus, in case a trace of  $\alpha; \beta$  is a successful trace of  $\alpha$  followed by a trace of  $\beta$ , it consists of first the internal steps of  $\alpha$ , then its external steps, followed by the external steps of  $\beta$ . The resulting trace has a number of internal steps followed by a number of external steps.

$\gamma = \alpha + \beta$ ,  $\alpha, \beta \in \mathcal{B}_i$ : Theorem 4.4 gives us that each trace of  $\alpha + \beta$  either is a trace of  $\alpha$  or a trace of  $\beta$ . Because of the induction hypothesis, all traces of  $\alpha$  and  $\beta$  consist of a purely internal trace followed by a purely external trace. All traces of  $\alpha + \beta$  thus satisfy the property to be proven.

$\gamma = \alpha \parallel \beta$ ,  $\alpha, \beta \in \mathcal{B}_i$ : All traces of  $\alpha$  and  $\beta$  consist of a sequence of internal transitions followed by a sequence of external transitions. We take two arbitrary terminating traces of  $\alpha$  and  $\beta$ , that is, traces that end in the empty statement  $\surd$  or in failure  $\not\rightarrow$ . We construct a terminating trace of  $\alpha \parallel \beta$  from these. By doing this construction in the most general way possible, we consider all terminating traces of  $\alpha \parallel \beta$ . Suppose we have the traces

$$\alpha_0 \xrightarrow{i} \alpha_1 \xrightarrow{i} \alpha_2 \xrightarrow{i} \dots \xrightarrow{i} \alpha_m \xrightarrow{p_1} \alpha_{m+1} \xrightarrow{p_2} \dots \xrightarrow{p_n} \alpha_{m+n}$$

and

$$\beta_0 \xrightarrow{i} \beta_1 \xrightarrow{i} \beta_2 \xrightarrow{i} \dots \xrightarrow{i} \beta_j \xrightarrow{q_1} \beta_{j+1} \xrightarrow{q_2} \dots \xrightarrow{q_k} \beta_{j+k},$$

with  $m, n, j, k \geq 0$ . Each of the two traces can either terminate successfully ( $\alpha_{m+n} = \surd, \beta_{j+k} = \surd$ ) or terminate in failure ( $\alpha_{m+n} \not\rightarrow, \beta_{j+k} \not\rightarrow$ ). As long as both  $\alpha$  and  $\beta$  still have internal steps to take, only the second and third transition rule for parallel composition apply. This rule interleaves the inter-

nal transitions of  $\alpha$  and  $\beta$  into a sequence of internal steps in the trace of  $\alpha\|\beta$ . After each step, there is a remaining statement  $\alpha_r\|\beta_s$ , where  $0 \leq r < m$  and  $0 \leq s < j$ , such that there is a choice between two internal next transitions  $\alpha_r \xrightarrow{i} \alpha_{r+1}$  and  $\beta_s \xrightarrow{i} \beta_{s+1}$ . When all internal steps of one the parallel branches have been done, this situation changes. Without loss of generality, we assume all internal steps of  $\alpha$  have been done. The remaining program then is  $\alpha_m\|\beta_t$ , with  $0 \leq t < j$ . In case  $\alpha_m$  fails, we have that  $n = 0$  and next transitions can only come from  $\beta_t$ . In case  $\alpha_m$  doesn't fail, the next step of  $\alpha$  is external, namely  $\alpha_m \xrightarrow{p_1} \alpha_{m+1}$ . The next step of  $\beta$  is  $\beta_t \xrightarrow{i} \beta_{t+1}$ , so now only the third transition rule applies and only  $\beta$  proceeds, until all of its internal steps also have been done and the remaining program is  $\alpha_m\|\beta_j$ . Up till now, the trace of  $\alpha\|\beta$  only contains internal transitions. Now, only the first transition rule is applicable, and  $\alpha$  and  $\beta$  simultaneously take external steps until one or both are finished. Without loss of generality, suppose  $\beta$  finishes. Termination is either successful ( $\beta_{j+k} = \surd$ ) or failing (with  $\beta_{j+k} \neq \surd, \beta_{j+k} \not\rightarrow$ ). In case  $\beta$  terminates successfully, the remaining statement is  $\alpha_u\|\surd$ , where  $m \leq u \leq m+n$ . This is equal to  $\alpha_u$ , and the remaining external steps of  $\alpha$  become the last steps of the trace of  $\alpha\|\beta$ . In case  $\beta$  terminates in failure, then so will the trace of  $\alpha\|\beta$ , as there is no transition rule applicable. Summarising, in all cases we get a terminating trace of  $\alpha\|\beta$ , consisting of a sequence of internal steps followed by a sequence of external steps. Non-terminating, partial traces always are a prefix of a terminating trace, and so these also have the desired property.

End of PROOF

Because agent programs basically are iterated behaviours, and we know that all traces of an iterated behaviour are concatenations of a number of traces of the behaviour iterated (Theorem 4.6), we now have conclusive evidence that the syntax and semantics of the skeleton programming give rise to agents performing cycles of reasoning and interacting.

Now that we have proven that agents executing behaviours first reason and then act, we look at two properties of parallel composition. Parallel composition as we syntactically and semantically defined it should be *commutative* and *associative*, which means that  $\alpha\|\beta = \beta\|\alpha$  and  $(\alpha\|\beta)\|\gamma = \alpha\|(\beta\|\gamma)$ , respectively. Here, the equality sign '=' should be read as 'is semantically the

same'. It would seem that  $\alpha = \beta$  if the set of all traces of  $\alpha$  is the same as the set of traces of  $\beta$ . We have a technical problem here, as we defined traces of a statement  $\alpha$  in Definition 4.20 to be sequences of transitions resulting from execution of the statement  $\alpha$ . Thus, a trace of  $\alpha$  never can be a trace of a different statement  $\beta$ , strictly speaking.

In order to express what it means for two composite statements to have the same semantics, we introduce *trace equivalence*. For the present purpose of proving associativity and commutativity, the changes to the agent state (mental state and sense buffer) and the actions taking place (labeled transition arrows) are relevant. To be precise, we call these traces of two different statements  $\alpha$  and  $\beta$  *equivalent*:

$\langle \sigma_0, \delta_0, \alpha_0 \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \alpha_1 \rangle \xrightarrow{l_2} \langle \sigma_2, \delta_2, \alpha_2 \rangle \xrightarrow{l_3} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \alpha_m \rangle$   
and

$\langle \sigma_0, \delta_0, \beta_0 \rangle \xrightarrow{l_1} \langle \sigma_1, \delta_1, \beta_1 \rangle \xrightarrow{l_2} \langle \sigma_2, \delta_2, \beta_2 \rangle \xrightarrow{l_3} \dots \xrightarrow{l_m} \langle \sigma_m, \delta_m, \beta_m \rangle$ ,  
because the internal and external actions done are the same and are in the same order. We call two sets of traces *equivalent* if for each trace in each set there is an equivalent trace in the other set.

In order to justify commutativity and associativity of parallel composition, we will reason along the same lines as we did in the case for parallel composition in the induction proof of Theorem 4.1. As ‘||’ can occur between two *Interactions*, two *Intractions* or two behaviours, and as  $Interactions_i \subset \mathcal{B}_i$  and  $Intractions_i \subset \mathcal{B}_i$ , we only have to look at the most general case, which is two or three behaviours in parallel.

**THEOREM 4.7 (Parallel composition is commutative)**

$\forall i \in \mathcal{I} \forall \alpha, \beta \in \mathcal{B}_i$  : the set of traces of  $\alpha||\beta$  and the set of traces of  $\beta||\alpha$  are equivalent.

**SKETCH OF PROOF:**

We take a trace of  $\alpha||\beta$  and show that there is an equivalent trace of  $\beta||\alpha$ . Analogously, we could show (which we won’t do) that for a trace of  $\beta||\alpha$  there is an equivalent trace of  $\alpha||\beta$ . Thus,  $\alpha||\beta$  and  $\beta||\alpha$  have equivalent trace sets.

So, take a trace of  $\alpha||\beta$ . It starts with a series (possibly empty) of internal transitions, followed by a series (also possibly empty) of external transitions. First, we look at the internal transitions. As the transition rules for parallel composition show, one internal transition in the trace originates from one of the two parallel statements,  $\alpha$  or  $\beta$ . Because the rules are symmetrical, we have that if  $\langle \sigma, \delta, \pi_1 || \pi_2 \rangle \xrightarrow{i} \langle \sigma', \delta', \pi'_1 || \pi_2 \rangle$ , then also

$\langle \sigma, \delta, \pi_2 \parallel \pi_1 \rangle \xrightarrow{i} \langle \sigma', \delta', \pi_2 \parallel \pi'_1 \rangle$ . We have an analogous property in case the step taken originates from  $\pi_2$ . This implies that the internal, reasoning sub-trace of the trace of  $\alpha \parallel \beta$  is equivalent to (the first part of) a trace of  $\beta \parallel \alpha$ .

Next, we look at the external transitions in the trace of  $\alpha \parallel \beta$ . The first transition rule for parallel composition shows that these external steps are created by summing up two external transitions originating from the parallel statements. As set union is commutative, we have that if

$\langle \sigma, \delta, \pi_1 \parallel \pi_2 \rangle \xrightarrow{i, (O_1 \cup O_2, C_1 \cup C_2, P_1 \cup P_2)} \langle \sigma', \delta, \pi'_1 \parallel \pi'_2 \rangle$ , then also

$\langle \sigma, \delta, \pi_2 \parallel \pi_1 \rangle \xrightarrow{i, (O_1 \cup O_2, C_1 \cup C_2, P_1 \cup P_2)} \langle \sigma', \delta, \pi'_2 \parallel \pi'_1 \rangle$ . So, from the external sub-trace of the trace of  $\alpha \parallel \beta$  in which both external statements from  $\alpha$  and  $\beta$  are performed, we can construct an equivalent external sub-trace which constitutes the second part of the trace of  $\beta \parallel \alpha$  we started on earlier. When there are no more external steps to be taken in one of the parallel branches, the external steps of the other branch complete the trace of both  $\alpha \parallel \beta$  and  $\beta \parallel \alpha$ .

We conclude that for the trace of  $\alpha \parallel \beta$  we chose there is an equivalent trace of  $\beta \parallel \alpha$ , and thus parallel composition is commutative.

End of PROOF

#### THEOREM 4.8 (Parallel composition is associative)

$\forall i \in \mathcal{I} \forall \alpha, \beta, \gamma \in \mathcal{B}_i$  : the set of traces of  $(\alpha \parallel \beta) \parallel \gamma$  and the set of traces of  $\alpha \parallel (\beta \parallel \gamma)$  are equivalent.

#### SKETCH OF PROOF:

We look at the way traces of  $(\alpha \parallel \beta) \parallel \gamma$  and  $\alpha \parallel (\beta \parallel \gamma)$  are constructed using the transition rules for parallel composition. As  $\alpha, \beta$  and  $\gamma$  are behaviours, they each give rise to a number of internal reasoning steps, followed by a number of external interacting steps. As we saw in the proof of Theorem 4.1, parallel composition of two behaviours interleaves the internal steps of two behaviours and sums up the external steps. As we now look at a parallel composition of two behaviours, one of which is another parallel composition of two behaviours, nothing new happens; we simply apply the transition rules for parallel composition twice for each transition of  $(\alpha \parallel \beta) \parallel \gamma$  and  $\alpha \parallel (\beta \parallel \gamma)$ . We start by looking at the internal steps of the resulting traces. For example, the first reasoning step of a trace of  $(\alpha \parallel \beta) \parallel \gamma$  can either be a first step of  $\gamma$  or a first step of  $(\alpha \parallel \beta)$ . In the latter case, a second choice has to be made, for either a first step of  $\alpha$  or a first step of  $\beta$ . This simply comes down to a choice between the three statements, and the exact same choice has to be made in order to find the first step of a trace of  $\alpha \parallel (\beta \parallel \gamma)$ .

Of course, this also applies to the other internal steps in traces of both parallel compositions. If the same choices are made along the reasoning phase of the execution of  $(\alpha\|\beta)\|\gamma$  and  $\alpha\|(\beta\|\gamma)$ , then the resulting internal sub-traces will be equivalent.

When the interaction phase is reached, then the third transition rule for parallel composition sums up the interactions with the environment from the three composite statements. As set union is associative, each step taken in a trace of  $(\alpha\|\beta)\|\gamma$  is also taken in a matching trace of  $\alpha\|(\beta\|\gamma)$ . When one of the three statements is finished, the other two keep taking external steps, until another statement is finished and the remaining statement takes its last external steps. The external sub-trace of a trace of one associative variant thus also has a matching, equivalent external sub-trace of the other associative variant.

For each trace of  $(\alpha\|\beta)\|\gamma$ , there is an equivalent trace of  $\alpha\|(\beta\|\gamma)$ , and vice versa. Thus, the trace sets of both associative variants are equivalent and parallel composition is associative.

End of PROOF

## 4.6 Illustration

The following example serves to illustrate how our model formalises several interaction aspects. Though the domain of the example is totally imaginary, similar interaction takes place in more realistic robotic applications, like in robot soccer.

Figure 4.3 pictures the scene. Three of the agents are busy doing the group action of skipping (in the sense of jumping over a swinging rope, and not of doing nothing). The fourth agent is skating towards the skipping agents on its

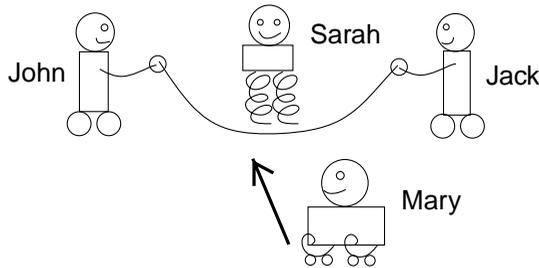


Figure 4.3: The robot playground

skeelers. The three agents skipping, John, Jack and Sarah, each have the group action SkipRope as part of their programs. In each group action scheme for this action, there are two individual actions SwingRope and one action JumpRope. The two SwingRope actions have to start at the exact same moment; otherwise, the rope won't move in the proper way for the jumping agent. The JumpRope action can start somewhat later, but not too late, as the jumping agent would get caught in the rope then. When the group action is executed, each agent chooses one action from a group action scheme, and starts doing its part. If they have chosen the same group action scheme, if each action from the scheme is executed by exactly one agent, and if the individual actions are synchronised properly, the group action potentially succeeds.

Whether the action SkipRope really succeeds depends on other actions and events taking place. In the figure, we see the fourth agent, Mary, moving closer on its skeelers. Mary is executing the individual action Skate. If Mary reaches the other agents, then the skipping action and the skeelering action will interfere and both will end in a clutter of falling robots. The skipping action can also be disturbed by an event. For example, the rope could break. In Figure 4.4, one possible scenario is depicted.

In this first scenario, the marks on the time lines of the agents indicate units of time, and the + and - signs indicate success or failure of the sub-action taking place during that time unit. Solid lines symbolise action parts that actually take place, and dashed lines show parts of actions that would have been done if there hadn't been a disturbance. Recall that global transition steps are computed by taking all sub-actions and events taking place during a time unit and computing the combined effects of these. Failing sub-actions also have effects, though these are disadvantageous.

Jack and John swing the rope perfectly simultaneously, and Sarah starts jumping over the rope one time unit after the rope began to swing. We assume this combination constitutes a valid group action scheme for SkipRope. Also, Mary is skating during a time interval which overlaps the interval of the skipping. So, interference of these two actions would be possible. But in this scenario, the disturbance is caused by an event, RopeBreaks. At the moment the rope breaks, the three individual actions contributing to SkipRope fail, and cannot continue any further.

In another scenario, depicted in Figure 4.5, the rope stays unharmed, but Mary bumps into Sarah. This means that the actions Skate and JumpRope fail, in time unit  $t_1$ . Jack and John still swing the rope at  $t_1$ , but their actions also fail in the next time unit, because the two robots lying on the floor obstruct the movement of the rope.

So, not all group action contributions immediately have to fail when one of

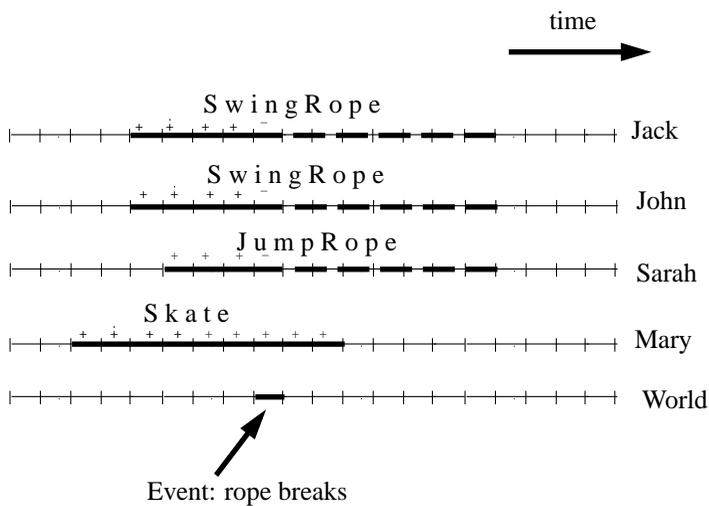


Figure 4.4: One scenario

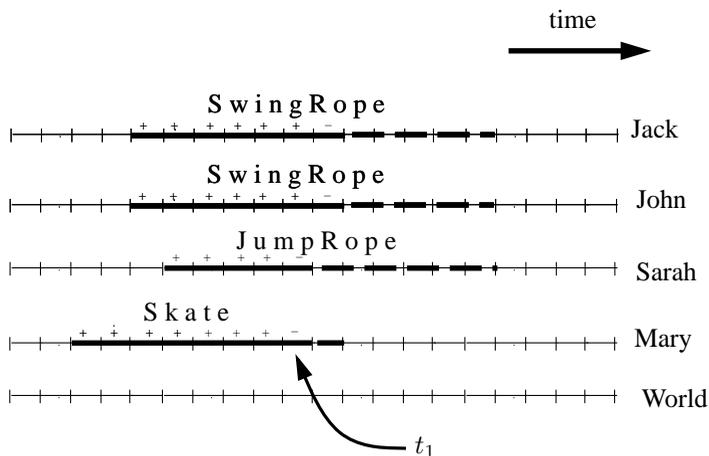


Figure 4.5: Another scenario

them does. It is even conceivable that one or more contributions are successfully terminated. For example, suppose Jack suddenly can't swing the rope anymore (his actuators need greasing). Then, the JumpRope action will fail in the same time unit, or a bit later, but John's SwingRope action can continue, in case Sarah moves away from the rope, such that she doesn't impede the movement of the rope.

## 4.7 Conclusions

In this chapter, we defined a real-time model of interacting agents situated in their environment, in which they execute cycles of internal reasoning and external interaction. We focus on faithfully modelling interaction of agents with each other and with their environment, including real-time aspects. We abstract from the inner functioning of the agents and let the programmer design the mental parts of the agents according to his own view of agents. The model consists of a skeleton programming language with its formal operational semantics. In order to obtain a usable programming language, several parameters of the language and the semantics have to be instantiated, namely:

### Domain parameters:

- \*  $\mathcal{P}$ , the set of propositional atoms used for formalising environment conditions
- \*  $\mathcal{I}$ , the set of agent identifiers
- \*  $\mathcal{A}_s$ , the individual actions
- \*  $\mathcal{A}_g$ , the group actions
- \*  $\mathcal{E}$ , the set of events
- \*  $\xi_i, i \in \mathcal{I}$ , the observability functions of the agents
- \*  $\tau$ , the world state transformation function

### Internal agent parameters:

- \*  $\mathcal{M}_f$ , the mental formula constructors applying to formulas
- \*  $\mathcal{M}_a$ , the mental formula constructors applying to actions

- \*  $v_i, i \in \mathcal{I}$ , the revision functions for the mental states of the agents, defining the change to the mental state when a mental formula is inserted or deleted
- \*  $\rho_i, i \in \mathcal{I}$ , the revision functions for the sense buffers of the agents, defining the change to the sense buffer when new observed and communicated information arrives
- \*  $\models$ , the consequence relation used for the internal agent state (consisting of mental state and sense buffer)

Domain parameters are common to many agent models, as modelling only one particular domain is not very fruitful. The internal agent parameters are more discerning, as in agent logics, architectures and programming languages it is often fixed which motivational and informational attitudes the agents use. By making the internal make-up of agents shapable by parameters of the skeleton programming language, we both add flexibility to our model and abstract away from details regarding internal reasoning. More specifically, it is the task of the system developer to make sure that mental formulas have a proper semantics and validate certain axioms. There are several ways to establish this, as both the revision functions for mental states ( $v_i$ ) and the consequence relation ( $\models$ ) are language parameters, and the system developer can program rules that ensure that axioms are always validated.

Another discerning feature of our model is that we have a strict separation between internal and external agent aspects. All operations an agent can perform either are *Intractions*, affecting only the internal agent state, or *Interactions*, influencing the state of the environment of the agent. This separation is also manifest in the reason–interact cycle, which is syntactically defined instead of semantically, as in other agent programming languages. A benefit of our reason–interact cycle is that it is fully programmable, while in other programming languages details of the processing of new information and the execution of commitments are fixed in the semantics of the language. We gave a proof that the syntactic restrictions on the skeleton programming language and the definition of the operational semantics indeed lead to the agents executing cycles in which internal reasoning precedes external interaction.

The semantical model we used for the skeleton programming language is inspired by step semantics, as used in [91]. It might seem that partial order semantics such as event structures [127] and Mazurkiewicz traces [87] also provide suitable models of true concurrency for our purposes. An event structure is a set of events (in the sense of actions of agents) with an associated partial order, representing causal dependencies between events. Events which

are independent can take place concurrently. Mazurkiewicz traces look different from event structures, but the underlying ideas are the same. An interesting issue in this context is that present partial order semantics usually are event-based, and thus don't model state transformations. As we are interested in computing the effects, that is the state transformation, resulting from a combination of actions and events, we would have to extend partial order semantics with states and state transformations. This has been done, as Van Eck relates in Section 7.2.1 of his thesis [39]. A global system state is described by the set of all events that have occurred in all processes (or agents). This set has to be closed with respect to causal dependency: if an event causes another event, and the second is in the global state, then so is the first.

The problem with the notion of state in partial order semantics is that it doesn't model in any straightforward way interference or synergy of events that are not causally dependent, and which take place simultaneously. For causally independent events, it doesn't matter which event is executed first, or not even whether the events take place during overlapping time frames. This is abstracted away in partial order semantics. In all cases, the outcome will be the same, because the events are independent. This view doesn't match with our aims for our model. In our view, actions and events don't have to be independent (non-interfering) in order to take place during overlapping time frames, as in partial order semantics. The outcome of interfering actions in our model is determined (among other things) by the timing of the actions. This idea contradicts the underlying ideas of partial order semantics, making them unsuitable as a candidate semantic model for the skeleton programming language.

In Section 7.2.1 of his thesis, Van Eck also argues that it is impossible or at least very hard to have global time available in distributed processes. At first sight, this claim seems to sweep away the foundation of our agent model, as we suppose there is a global clock, at the clock ticks of which the agents interact. Van Eck states that in order to establish global time, either there must be a single source of time that provides the clock signals of all processes (that is, agents), or the processes have different local clocks, which must be synchronised. A central clock is not feasible when systems are physically distributed, and the time signal is delayed in different ways during transportation. Moreover, synchronisation of local clocks is very difficult, according to Tanenbaum (pages 471–476 in [116]). Tanenbaum states that getting the local clocks synchronised to within 5 or 10 msec is expensive and hard. We can rebuke these arguments for our framework, because we study common sense agent behaviour. Therefore we operate on a different level of abstraction than the level Van Eck describes, which is the level of processor clocks and atomic processor actions. We take a higher-level viewpoint, at which the length of the time unit is much

larger than a clock tick of an agent processor. So, synchronisation to within 5 or 10 msec would be enough for our purposes.

Our model of interacting agents incorporates features that seem to be missing in other approaches to agents:

- \* A dynamic world, in which events can occur. We realise this through the incorporation of a separate world state, in which changes can occur without some agent causing them (events).
- \* Actions of observation, which agents perform to obtain an internal representation (in their sense buffers) of the state of the world. In other approaches, sensing the environment is done implicitly, resulting in an internal database which is always consistent with the world state. In our approach, the agents need to observe and communicate to maintain accurate information.
- \* Actions with duration. Both group actions and individual actions take a number of time units. So, interaction of several actions happening during overlapping time frames can be modelled in a natural way. A real-time model of action execution is incorporated in the semantics of the skeleton programming language. In the global semantics, two actions of different agents could interfere, yielding results different from the results of some interleaving of the actions.
- \* The choice between group actions and individual actions. Agents can coordinate some of their actions while doing other actions by themselves. Group actions and individual actions are related through group action schemes, which specify the individual actions the group members contribute to the group action and the way these individual actions have to be synchronised.

Thus, we have created an operational, abstract model of interacting agents, incorporating real-time aspects. As our model is abstract, it is possible to compare several agent systems constructed using different instantiations of our abstract model.

Other approaches of agent programming generally abstract from some or all of the issues mentioned above. This yields agent models in which important problems can't be studied. Because of the presence of these features, our model allows a more realistic view on the behaviour of diverse systems of agents situated in dynamic environments.



---

## An Abstract Programming Language for Agents Interacting through Group Actions

---

*If we stopped talking in circles we might  
get closer to the earth*

*Hothouse Flowers*

### 5.1 Introduction

In this chapter, we introduce the syntax and semantics of a constraint programming language for multi-agent systems with facilities for group communication, group formation and group collaboration. The language is called GrAPL, which abbreviates Group Agent Programming Language. Groups in GrAPL are dynamic, and can be created at runtime. Coordination and cooperation, which are crucial notions in multi-agent systems, are modelled in GrAPL by means of formally defined primitives for dynamic group communication and synchronisation.

One of the central ideas of the agent paradigm is that there is added benefit in having multiple, autonomous agents in a system. The strength of this is that these agents can pursue their own goals, but also form alliances with other agents when this is necessary or more beneficial than individual activity. As we stated in earlier chapters, *interaction* is at the heart of the agent concept. In this chapter, we focus on interaction between agents that form groups to negotiate about and execute group actions, and we leave interaction between an agent and its environment out of consideration.

While coordination of agents is crucial in multi-agent systems, proper formalisations in agent programming languages nevertheless are scarce. Formalisations do exist in other areas of agent research. If we divide the work done on agents into agent *theories*, *architectures* and *languages* (the classical ATAL categories), then we find that collaboration and coordination of groups of agents are studied in agent theories and in agent architectures, while this isn't happening so much in agent programming languages. Existing agent theories logically describe mental attitudes held by groups of agents, like joint beliefs and intentions [83]. Also, theories about commitments are proposed to formalise group agreements that influence future actions of the agents participating in the group [33, 103, 114]. In the area of agent architectures, there are various proposals for architectures for multi-agent coordination [67, 118]. Jennings developed GRATE\* [67], which is a layered architecture. One of the layers takes care of cooperation and control. The architecture is heavily inspired by logical theories of beliefs, desires, and (joint) intentions. Tambe proposes STEAM [118], which is an architecture for flexible teamwork, where shared partial plans are executed by dynamic teams. But in agent-oriented programming languages, formally defined statements for group communication and synchronisation are novel.

The language GrAPL fits in the tradition of the programming languages ACPL [12] and 3APL [59, 60]. Each of these languages focuses on a different aspect of agency: ACPL contains statements for agent communication, 3APL focuses on the internal practical reasoning of agents trying to reach their goals, and GrAPL offers primitives for group formation and group action. GrAPL provides functionality not present in 3APL (which is mainly single-agent oriented) and ACPL (which isn't focused at coordinating actions). We almost exclusively focus on group negotiation and group action, while ACPL is constructed for general agent communication. We don't devote attention to the effects of actions on the environment and the beliefs of the agents, as this is already covered in 3APL. In this sense, GrAPL is an *abstract* programming language; we primarily developed novel statements for group formation, group communication and group action execution, and didn't put in much effort for

the other agent abilities, such as ordinary communication, observation and practical reasoning. GrAPL is not meant to be yet another agent programming language; ultimately, we aim at a language incorporating all aspects of agency.

We first give an informal explanation of the main ideas underlying the new features of the programming language GrAPL. The basic idea of GrAPL is that agents synchronously communicate in order to dynamically form groups which synchronously perform certain actions. The agents negotiate about *constraints* on these group actions. The constraints are logical formulas, prescribing properties of the action parameters, e.g., the time and place of a meeting, and the group of participants of the meeting. Subsequently, several agents can synchronously execute the action, obeying the constraints associated with the action.

The programming language GrAPL is based on the paradigm of constraint-based reasoning which enjoys much interest in artificial intelligence because it has proven to be a powerful method for solving complex problems like planning, resource allocation and scheduling. Our programming language can be implemented on top of a *constraint solver* [119], which finds a consistent assignment of the action parameters satisfying the constraints of a group of agents. GrAPL incorporates ideas from concurrent constraint programming [111] to enable agents to produce and query constraints about the parameters of the group actions. An important difference is that each agent has a local constraint store for each action, while in constraint programming there usually is one global store.

Another inspiration for GrAPL comes from synchronisation mechanisms in concurrent programming, more specifically from synchronous communication in CSP [61]. In this language, synchronous communication is used to communicate values. The principles of communication in GrAPL are similar, although communication is not bilateral but multilateral, and formulas are communicated instead of values. We opt for synchronous communication instead of asynchronous communication, because using synchronous statements, agents can't communicate a constraint to other agents if these other agents aren't also communicating with them. This way, the agents always control whether information is written in their constraint stores. By using established ideas from concurrent programming and constraint solving, we hope to prevent reinventing the wheel. Although multi-agent systems have their unique features, many aspects of agents have been studied in other fields of computer science, and results can be adapted to fit multi-agent systems.

More specifically, the communication and coordination process which should lead to the synchronous execution of an action by a group of agents, distinguishes two phases. During the first phase, called the *negotiation phase*,

groups of agents negotiate about the constraints they impose on a certain group action  $a$  by synchronously communicating their constraints on the action parameters. All actions have an implicit parameter which denotes the group involved. By means of this parameter agents may express their constraints on the composition of the group. Group communication updates the constraints of the participating agents on the action  $a$  to be the conjunction of the individually proposed constraints. Subsequently, this resulting formula constrains each execution of  $a$  for each agent that has participated in the communication, until the constraint on the action  $a$  is changed again. In this second phase, called the *execution phase*, a group of agents tries to synchronously execute a group action which was the subject of negotiation. If the actual parameters of all agents in the group are compatible, and the constraints of the agents allow these actual parameters, the action is executed by the group. Otherwise, group action execution fails. The constraints thus *monitor* the execution of the action.

In Section 5.2 we give an intuitive account of the new constructs in GrAPL. In Section 5.3 we introduce the syntax of GrAPL. Section 5.4 describes the formal operational semantics. In Section 5.5, we illustrate the new features of GrAPL in three extensive examples. Section 5.6 concludes this chapter.

## 5.2 Intuitions

Before going into the formal details of the syntax and semantics of the programming language, we will give an intuitive sketch of the new features of the language. As stated in the introduction, the focus of this work is group formation, group communication and group action. First, agents synchronously communicate with each other in order to form groups that are committed to performing group actions together. The agents communicate about constraints on the action they might do together. Subsequently, several agents can synchronously execute the action, obeying the constraints associated with the action. Not all actions need to be constrained, but if a set of constraints is associated with an action, then action execution has to obey the constraints, which can specify demands on the parameters of the action and the group of agents participating in the action. In other words, the constraints monitor the execution. To facilitate this, there are three special statements in the programming language. The first two implement group communication (`CommGroupAdd` and `CommGroupReset`) and the third implements (group) action execution (simply  $a(t_1, \dots, t_n)$ , where  $a$  is an action and  $t_1, \dots, t_n$  are the actual parameters of the action).

Actions in GrAPL are parameterised. Each action has a certain arity, which

is the number of parameters it needs. An example of an action of arity two is  $\text{PlayGame}(v_1, v_2)$ , where the first parameter  $v_1$  is the particular game to be played and  $v_2$  is the starting time of the game. Apart from these explicit parameters, each action has one implicit parameter, which is the group of agents performing the action. This group is always denoted by the special variable  $g$ .

In the introduction, we mentioned the two phases of the coordination process for a group action. In the first phase, the negotiation phase, the agents interested in establishing conditions for execution of a group action  $a$  perform  $\text{CommGroupAdd}$  and  $\text{CommGroupReset}$  statements, thereby exchanging proposals for constraints on the action  $a$ . As the name of the statements already suggests,  $\text{CommGroup}$  statements (both kinds) are synchronously performed by a group of agents. In this respect,  $\text{CommGroup}$  execution is similar to the synchronous communication primitives in (for example) CSP [61]. A difference with CSP is that we have multi-party synchronisation, while in CSP only two processes participate in synchronous communication. The constraints the agents communicate about in GrAPL are constraints on the explicit and implicit parameters of  $a$ . So, agents talk about the details of the action and about the group which is going to perform the action.

Each agent has a private, local constraint store, where it keeps the present constraint on each action. We could also have opted for a global constraint store for each action, in which agents communicating about the action write their constraints. Local constraint stores have several advantages over global stores:

- \* They fit better with the concept of autonomous agents. The intuitive idea of an agent is that it reasons about its own motivations and the circumstances in its environment, and then decides to do certain actions, by itself or in cooperation with other agents. All information the agent needs for this is stored locally (in its mental state) or is received from the environment (through observation and communication). As the constraints in the constraint store describe information about how future actions need to be executed, this information fits well in the local agent state. As another argument, a multi-agent system generally consists of a number of agents, situated in an environment. Global constraint stores aren't agents, and thus their presence as separate entities in a multi-agent system is counterintuitive, from the agent-oriented viewpoint.
- \* They allow multiple groups of agents to discuss and execute the same action. When two disjoint groups of agents negotiate on an action  $a$ , then the local constraint stores of the agents in each group contain the

constraints which the agents in this group communicated, and no constraints coming from the agents of the other group. In case there is a global constraint store, the constraints of both groups would arrive in the same global store, making separate negotiations impossible. With local constraint stores, the two groups can execute the action  $a$ , resulting in two separate executions of instances of  $a$ , each with different actual parameters, which fit the constraints of the two groups involved.

- \* The local constraint store of an agent represents its own view on the negotiation about a certain action. During each synchronous `CommGroup` meeting, only communicating agents update their local stores. So, in order to know what's going on, it is essential to be present in meetings, like in many real-world organisations. The group of negotiators which communicates about a certain action instance doesn't have to be fixed. Agents can leave the group, and others can join it, when the constraints on the group composition allow this. When agents leave, they aren't informed anymore about new demands on the action. They can reset their constraint store, and start negotiation with another group.

Local constraint stores thus match the intuitions associated with agents. Initially, the local constraint stores contain the constraint  $\top$ , which denotes the logical formula that is always true. So, initially no agent has any demand on any action.

`CommGroupAdd` and `CommGroupReset` both take two arguments, a constraint  $\varphi$  and an action  $a$ . The difference between `CommGroupAdd`( $\varphi$ ,  $a$ ) and `CommGroupReset`( $\varphi$ ,  $a$ ) is that in the first case the agent adds  $\varphi$  to its present constraint on  $a$ , and then proposes the conjunction, while in the second case, the agent forgets about its current constraint on  $a$  and simply proposes  $\varphi$ . When a group of agents is communicating about a group action, some agents in the group may perform a `CommGroupAdd` action and others may perform a `CommGroupReset` action. All agents in the group have to agree upon the focus of the communication, that is, the action. Each agent brings its own set of constraints, demanding execution of the action to take place in a certain manner. The composition of the group of communicators must satisfy the demands of each agent. If this is not so, group communication fails. Each successful synchronous combination of group communication actions updates the constraints of the agents communicating on the action discussed to be the conjunction of the proposals, as this is the weakest constraint implying all individual constraints.

After one or more synchronous executions of `CommGroupAdd` and `CommGroupReset` statements, the second phase, the execution phase, takes

place. In the execution phase, a group of agents tries to execute the group action which was the subject of negotiation. Each agent  $\iota$  chooses actual parameters for the action  $a$ , named  $t'_1, \dots, t'_n$  if the action takes  $n$  parameters. As the parameters can contain free variables, certain agents in the group can communicate actual parameters to other agents. This implicit communication which takes place as a side-effect of action execution is called *execution-time communication*. We will show how this works in Example 5.1 below. If the actual parameters of all agents in the group are compatible, and the constraints of the agents allow these actual parameters, the action is executed by the group. So, the second ‘new’ element in our language (next to group communication) is group action execution. As can be seen above and in Example 5.1, the syntax of action execution is conventional; it’s the semantics that is different. Whenever an action  $a(t_1, \dots, t_n)$  is to be executed by an agent, the abstract interpreter of our language checks whether there is a constraint bound to this action. This is done for all agents about to perform the action. There must be no conflicts in the actual parameters and all constraints in constraint stores of agents participating have to be satisfied. If so, the action can be done. Otherwise, group action execution fails.

Each agent has its own belief base, which it uses to store information. The beliefs of an agent can influence the decision making of the agent, for example about the constraints the agent imposes on some action. When designing GrAPL we didn’t focus on sophisticated mental agent processing, as this is already done in other agent programming languages. We abstract from the agent’s mental processing, and provide two basic operations on the belief base, namely insertion and testing. In this chapter, we reserve the term *constraint store* for sets of formulas which describe demands on parameters of certain actions. In constraint programming languages, like for example CCP ([111]), this term covers all information stores. Our use of the term is different; in our view, belief bases need not be constraint stores. Belief bases can be constraints, describing the partial information of the agents about certain world features (represented by variables), or they can be sets of closed formulas, describing the information which the agents hold true of the world state. Because we choose for the second option in this chapter (which isn’t a principled choice in any way), we only refer to constraints on actions with the phrase ‘constraint store’.

As a first example of the constructs of GrAPL, we return to the game playing action.

### EXAMPLE 5.1 (Playing a game)

Suppose we have agents with names Gabriël, Jan-Ybo, Martha and Jantina. It is a boring Sunday afternoon, and the agents are talking about playing a game. Initially, all four agents have the empty constraint ( $\top$ ) on `PlayGame` in their local store. They enter the negotiation phase. The four agents synchronously perform a `CommGroupReset( $\varphi_{id}$ , PlayGame)` action. Here,  $\varphi_{id}$  is the constraint of the agent named  $id$ , as shown below. Recall that the action `PlayGame`, introduced above, needs two parameters, which we refer to as  $v_1$  and  $v_2$ . As will be explained later, in constraints all agents always use formal parameter  $v_i$  to refer to the  $i$ th parameter of some action. Each of the agents proposes a different constraint:

- \*  $\varphi_{Jantina}$  is the following constraint:  $v_1 = \text{Rummikub} \wedge v_2 < 16.00$ . So, the only game Jantina is prepared to play is Rummikub, and she wants to start before four o'clock. Jantina has no demands on the composition of the group playing the game.
- \*  $\varphi_{\text{Gabriël}}$  is this constraint:  $v_1 = \text{Rummikub} \rightarrow \text{Martha} \in g$ . So, Gabriël is willing to play any game at any time, but if the game is to be Rummikub, he wants Martha to play along.
- \*  $\varphi_{\text{Jan-Ybo}}$  is the constraint  $v_2 > 14.30$ . His belief base contains the information that his favourite TV-show is on from 13.30 till 14.30 and that he has to do his homework up to 13.30. So, Jan-Ybo has other things on his mind until half past two. Only at some later time he is prepared to play a game.
- \*  $\varphi_{\text{Martha}} = \top$ . Martha is a very easy agent. She doesn't have any constraints.

These four constraints are consistent; the conjunction of the constraints is equivalent to  $v_1 = \text{Rummikub} \wedge 14.30 < v_2 < 16.00 \wedge \text{Martha} \in g$ . Consequently, the four individual local constraints are replaced by the constraint just mentioned.

Now, after the negotiation phase, which in this case consists of only one synchronous group communication action, the four agents can actually play the game. They enter the execution phase. Each of the four agents executes a `PlayGame` action. The actual parameters for this action may differ, as long as they can be unified and they obey the constraints agreed upon. The agents can test their constraint stores for Rummikub to find out which demands the other agents communicated, and to choose appropriate actual parameters. We don't go into this now.

Some actual action parameters can be free local variables. This way, execution time communication about action parameters is possible. This form of communication is a side-effect of action execution. Suppose this is what the four agents try to do:

- \* Jantina:  $\text{PlayGame}(\text{Rummikub}, t_1)$ . So, Jantina doesn't provide a value for the starting time of the game. Instead, she uses a free local variable, denoting that she doesn't want to be the one to choose the definite time, even though she initiated the constraint that the game has to start before 16.00.
- \* Gabriël:  $\text{PlayGame}(p_1, t_2)$ . Gabriël is open to anything (as long as the constraints are satisfied). He simply uses two free local variables as actual parameters.
- \* Jan-Ybo:  $\text{PlayGame}(\text{Rummikub}, 15.00)$ . So, Jan-Ybo sets the time at which the game will start.
- \* Martha:  $\text{PlayGame}(p_2, t_3)$ . Martha still is a very easy agent. She is prepared to adjust herself fully to the other agents, as long as the constraints communicated to her in the first phase are respected.

An attempt to synchronously execute this group action will succeed, as the concrete parameters of the four agents can be unified, and the resulting action parameters satisfy the constraints. Jantina and Jan-Ybo agree on the first parameter of the action, and they communicate Rummikub to the other agents. The second parameter is picked by Jan-Ybo to be 15.00, and implicitly communicated to the others. These two actual parameters satisfy the demands made on the formal parameters  $v_1$  and  $v_2$  in the (now identical) constraint stores of the agents. Also, Martha is part of the group playing the game, and so the demand on the group composition ( $\text{Martha} \in g$ ) also holds.

But in case Jantina would have demanded the time 14.45 as the second parameter, the group action would have failed. In this case, there are two agents in the group who don't agree on a parameter and execution time communication fails. A group action can also fail when the agents agree on the actual parameters. This happens when the unified parameters resulting from execution time communication conflict with the constraints of the participants. For example, Jan-Ybo can choose the second actual parameter of  $\text{PlayGame}$  to be 17.00, and tell this to the other agents using execution time communication. Agent Jan-Ybo then chooses to ignore the constraint on  $\text{PlayGame}$  that the agents agreed upon in the negotiation phase. As the other agents use a free local variable for the second parameter, execution

time communication succeeds, but the action can't succeed, as the time 17.00 doesn't obey the constraint on `PlayGame`. When a group action fails, the agents can wait until the action can be successfully executed, or they can choose to continue doing other actions from their programs.

The picture sketched above just gives a general impression of the features of the programming language. There are many subtle issues and different interesting options for the precise meaning of group communication and coordination in this language. We will elaborate on this later on in this chapter.

## 5.3 Syntax

### 5.3.1 Basic sets and conventions

The programming language we are about to define is based on the principles of constraint programming. The sets underlying GrAPL are:

- \*  $\mathcal{A}$  = the set of atomic actions. Typical elements are `a` and `b`. Each action has an arity, which is the number of parameters it takes.
- \*  $\mathcal{I}$  = the set of agent identities. Typical elements are `l` and `κ`.
- \*  $\mathcal{V}$  = the set of variables. There are two kinds of variables;  $\mathcal{V} = \mathcal{LV} \cup \mathcal{GV}$ . Here,  $\mathcal{LV}$  are local variables. Each agent has its own local variables, so the set of local variables is the disjoint union of sets of agent-specific local variables.  
 $\mathcal{GV}$  contains the global (system) variables, defined as  
 $\mathcal{GV} = \{v_k | k \in \mathbb{N}\} \cup \{g\}$ .
- \*  $\mathcal{D}$  = the value domain. Elements of this set are used as constants and can be bound to variables. Two subsets of  $\mathcal{D}$  are  $\mathcal{I}$  and  $\wp(\mathcal{I})$ .

Local variables are used for processing information only relevant to one agent, such as testing the local constraint store and specifying formulas to be inserted into the belief base. For this last purpose, the programmer must make sure that free local variables are instantiated with ground values at the time the insertion is executed, as the belief base is a closed formula. Local variables are also used to specify actual action parameters, when an agent doesn't care about the precise value of certain formal parameters. Above, we attributed disjoint sets of local variables to all agents. These disjoint sets make it impossible

for two agents in a system to use the same local variable name. This prevents name clashes, which could occur during synchronised action. In group action execution, the local variables of a number of agents meet when there is execution-time communication. The following example illustrates the potential problems:

EXAMPLE 5.2 (Local variables clash)

Suppose two agents  $ag_1$  and  $ag_2$  attempt a group action  $a$ , having three parameters. Agent  $ag_1$ 's program contains the statement  $a(x, 4, 58.3)$  and  $ag_2$  has the statement  $a(1, x, 58.3)$ . To come to a successful execution of this group action, we want to find a global substitution which unifies the actual action parameters. The local variable  $x$  of  $ag_1$  has to be bound to the value 1, while the variable  $x$  of  $ag_2$  has to be bound to 4. Globally (at the system level), these two value associations clash, as there exists no unifying overall substitution doing the trick. But when we look at the action statements, it is clear that the group action can be executed successfully: the common first parameter has to be 1, the second 4 and the third 58.3. As soon as one of the agents replaces  $x$  with  $y$ , a unifying global substitution does exist.

In order to prevent these name clashes, we define the sets of local variables in such a way that each agent uses different local variables.

$\mathcal{GV}$  is the set of all variables  $v_i$  and  $g$ . We use these variables to specify the formal parameters of actions. We adopt the practice to refer to these parameters in a uniform manner. We call the formal parameters of an action  $a$   $v_1, v_2, \dots, v_k$  if the arity of  $a$  is  $k$ . The implicit formal parameter for the group performing the action is  $g$ . In GrAPL-programs, these formal action parameters occur in constraints on actions and in formulas tested on the constraint stores. All agents use the same set of global variables to refer to formal parameters of the whole range of actions. Constraints are always specified relative to an action, so the global variables have an unambiguous meaning. Thus, the set  $\mathcal{GV}$  exactly contains the variables needed to work with constraints. To avoid confusion, global variables are only allowed in conjunction with constraints. Also, global variables are never bound to values. Even if  $v_1 = 7$  is a demand on an action  $a$ ,  $v_1$  isn't bound to 7; the constraint just means: "The first parameters of  $a$  must be 7", it doesn't mean that the variable  $v_1$  *always* has the value 7. We don't generate bindings to global variables because they are used in constraints on different actions, and it is undesirable that a constraint like  $v_1 = 7$  on *one* action demands the first formal parameter of *every* action to be 7. Because global variables aren't bound to values, there can be no value clashes between

agents involving global variables. We chose to introduce special global constraint variables because it simplifies communicating about action parameters by groups of agents. If an agent is communicating with other agents about constraints on some action, it doesn't have to indicate which names it uses to refer to the different parameters of the action. Each agent uses the convention of referring to the  $i$ th parameter by  $v_i$ . So, no unification is necessary and programs become more clear.

We stress that our use of the terms 'local' and 'global' may be different from use elsewhere. In many programming languages, the terms 'local' and 'global' are meant relative to pieces of program code. Here, we have a different usage. 'Local' means private to one agent, and 'global' means shared by all agents in the multi-agent system. Global variables are used in communication and group actions, which naturally involve a number of agents, while local variables are used for operations concerning one agent only.

GrAPL makes use of a multi-sorted predicate logical language  $\mathcal{L}$ . Each agent possesses a *belief base*; this contains closed formulas (no free variables) from  $\mathcal{L}$ . The constraints on actions are also formulas from  $\mathcal{L}$ , prescribing properties of action parameters. Each agent locally stores the present constraint for each action. We elaborate on  $\mathcal{L}$  now.

**DEFINITION 5.1** (The logic  $\mathcal{L}$ )

$\mathcal{L}$  is a multi-sorted predicate logic. The set of variables of the logical language is  $\mathcal{V}$  and the set of constants is  $\mathcal{D}$ . The logic includes set theoretic predicates and functions, such as  $\in, \subseteq, \cup$  and  $\cap$ , to express properties of the composition of groups of agents, as well as predicates and functions to describe properties of action parameters. We use  $\varphi$  and  $\psi$  to denote arbitrary formulas from  $\mathcal{L}$  and  $\top$  and  $\perp$  to denote the formulas that are always true and false, respectively.

We denote the set of free variables in an expression, term, formula, program, or other syntactic form  $w$  by  $free(w)$  and the set of all its variables by  $var(w)$ .

We assume the logic  $\mathcal{L}$  is equipped with an entailment relation, denoted by  $\models$ .

As an example of the use of  $\mathcal{L}$  for formulating constraints, we give some constraints on the action  $MoveObject(v_1, v_2, v_3)$ . Here, the first formal parameter is the object to be moved, the second parameter is the original location of the object and the third parameter is the location to which the object has to be moved. A very simple constraint is:  $v_1 = table$ . If an agent has this constraint on action  $MoveObject(v_1, v_2, v_3)$ , then it is only willing to move the table; any attempt

of this agent to move something else will fail. If the predicate logic contains a function `distance` which takes two locations and yields the distance between these two locations, then another simple constraint is  $\text{distance}(v_2, v_3) < 10$ . An agent having this constraint associated with the `MoveObject` action, is not prepared to move something over a distance which is 10 or greater. A last example of a simple constraint is  $\text{James} \in g$ . This means that the agent having this constraint is only prepared to move things when James is part of the group performing the `MoveObject` action.

By using logical operators, more complex constraints are obtained. An example is the constraint

$$\text{distance}(v_2, v_3) \geq 20 \rightarrow (v_2 = \text{Utrecht} \wedge \text{Max} \notin g) \vee \#(g) > 5,$$

which states that when the distance an object has to be moved over is 20 or more, the agent having this constraint only agrees to help if there are at least five agents cooperating or if the moving starts in Utrecht and the agent doesn't have to cooperate with Max.

If a certain constraint is associated with an action `a`, then this constraint monitors future executions of `a`. Agents can repeatedly communicate with each other, using `CommGroupAdd( $\varphi$ , a)` and `CommGroupReset( $\varphi$ , a)`. By doing this, the agents define the parameter space of the actual parameters of `a`. When agents perform `CommGroupReset` statements, then their local constraint stores are re-initialised with fresh constraints, and when agents add constraints using `CommGroupAdd`, they narrow down the possible values for the formal parameters of the action.

Group communication can be used by the agents for different purposes. By performing `CommGroup`-statements, the agents can communicate their preferences (desires), or take care that the group action is executed in such a way that their objectives (goals) are reached. Another view on group communication is that agents establish social norms on the permissible action parameters and enforce proper societal behaviour this way. Group communication can also be used to exchange information on the precondition of a certain action execution. An action may be executable with all possible parameter sets, in which case constraints of a group of agents only display their personal preferences or their group norms, not their knowledge on executability of the action. On the other hand, it is possible that the agents have to discover the environment conditions that allow action success. In this case, the action has a precondition, which is only partly known to the agents. So, the agents have to cope with the well-known qualification problem ([88]). Using `CommGroup` statements, they can exchange information on the precondition in a cooperative manner. The constraints on the action parameters narrow down the actual parameter space. So, the precondition of the action is strengthened by group communication. The

postcondition isn't directly affected, but as the action can only be executed with certain parameter sets, the set of possible resulting world situations will also be narrowed down.

Actions are primitive notions; their meaning and effects are laid down in semantic functions. Nevertheless the agent program can influence the meaning of actions, because agents can constrain the set of permissible actual parameters by group communication.

### 5.3.2 Syntax of programs

We denote the set of agent programs by  $\mathcal{P}$ . In order to define this set, we first define the set of *basic statements*  $\mathcal{S}$ .

DEFINITION 5.2 (Basic statements)

The set  $\mathcal{S}$  of basic statements is the smallest set containing:

- \* skip
- \*  $?\varphi$ , where  $\varphi \in \mathcal{L}$  and  $\text{var}(\varphi) \cap \mathcal{GV} = \emptyset$ .
- \*  $?( \varphi, \mathbf{a} )$ , where  $\varphi \in \mathcal{L}$  and  $\mathbf{a} \in \mathcal{A}$ .
- \*  $\text{ins}(\varphi)$ , where  $\varphi \in \mathcal{L}$  and  $\text{var}(\varphi) \cap \mathcal{GV} = \emptyset$ .
- \*  $\text{CommGroupAdd}(\varphi, \mathbf{a})$ , where  $\mathbf{a} \in \mathcal{A}$  and  $\varphi \in \mathcal{L}$ .
- \*  $\text{CommGroupReset}(\varphi, \mathbf{a})$ , where  $\mathbf{a} \in \mathcal{A}$  and  $\varphi \in \mathcal{L}$ .
- \*  $\mathbf{a}(t_1, \dots, t_k)$ , where  $\mathbf{a} \in \mathcal{A}$ , the arity of  $\mathbf{a}$  is  $k$  and all  $t_i$  are terms of  $\mathcal{L}$ , such that for all  $i \in \{1, \dots, k\} : \text{var}(t_i) \cap \mathcal{GV} = \emptyset$ .

The language includes a statement for doing nothing, skip.

There are two kinds of tests, namely tests of the belief base (simply denoted  $?\varphi$ ) and tests of the constraint bound to an action (denoted by  $?( \varphi, \mathbf{a} )$ ). These tests check whether the formula  $\varphi$  is logically entailed by the belief base or the current constraint on  $\mathbf{a}$ , respectively. Both kinds of tests can yield bindings of values to variables, but these variables never are global variables. For tests of the belief base, this is achieved syntactically, by forbidding global variables in the formula tested. Because global variables are used by all agents to refer to the formal parameters of all actions, it would be unpractical to generate bindings to global variables. Besides the practical reason of making no bindings to global variables, there also is a conceptual reason for excluding global variables from some statements. This reason is that we introduced global variables specifically for constraint handling. Therefore, they are forbidden in formulas tested on and inserted into the belief base, and also in

actual action parameters (this is what is meant in the definition above by for all  $i \in \{1, \dots, k\} : \text{var}(t_i) \cap \mathcal{GV} = \emptyset$ ). This way, we maintain a clear separation between global and local processing, which adds clarity and elegance.

In tests of actions, we do allow global variables, as the constraint on the action can contain global variables. For example, suppose an agent has the constraint  $v_1 \leq 10$  on action  $\mathbf{a}$ , meaning that the first parameter of  $\mathbf{a}$  must not be larger than 10. The test  $?(v_1 = 10, \mathbf{a})$  tests whether the constraint on  $\mathbf{a}$  implies that the first parameter of  $\mathbf{a}$  is 10. As this is not a logical consequence of  $v_1 \leq 10$ , the test fails. In case the current constraint on  $\mathbf{a}$  would have been  $v_1 = 10$ , then the test succeeds. The semantics of GrAPL is defined such that this successful test doesn't result in a binding of the value 10 to the global variable  $v_1$ . In case we would perform  $?(v_1 = x, \mathbf{a})$ , where  $x$  is a local variable, and the constraint on  $\mathbf{a}$  implies that  $v_1 = 10$ , then only the local variable  $x$  is bound to 10.

The statement  $\text{ins}(\varphi)$  adds the information  $\varphi$  to the belief base. As mentioned above, we want global variables only to be used in conjunction with constraint stores, so we forbid them to occur in new belief base formulas. Another issue is that free local variables are not allowed in new belief formulas, as the belief base has to be a set of *closed* formulas. But we won't demand that  $\text{free}(\varphi) = \emptyset$ , because this would mean that each belief inserted into the belief base must already be completely specified at compile time. So, we allow free local variables in the new belief formula  $\varphi$ , but we demand that each free variable is *guarded* by a preceding test or action execution, which yields a value for this variable. (Note that action execution can generate bindings, through execution time communication.) Here is a simple example to make matters clear:

EXAMPLE 5.3 (Guarded statements)

```
?weight( $x$ ) < 10;
WalkTo(supermarket);
ins(light( $x$ ))
```

In this peculiar program, where an agent is thinking about light objects while walking to the supermarket, there is an insertion into the belief base which contains the free variable  $x$ . But this free variable is bound by the test on the belief base, which takes place earlier. So, the execution of the  $\text{ins}$ -statement is guarded.

The most innovative statements of the programming language are  $\text{CommGroupAdd}(\varphi, \mathbf{a})$  and  $\text{CommGroupReset}(\varphi, \mathbf{a})$ . Here,  $\mathbf{a}$  is the action the agent communicates about and  $\varphi$  is a constraint stating demands of the agent on

future executions of the action  $a$ . Using these statements, agents synchronously communicate about the details of the action and about the group which is going to perform the action. Each agent in a group of communicators executes either a `CommGroupAdd`-statement or a `CommGroupReset`-statement. Arbitrary combinations of these two statements are allowed. Group communication succeeds if every agent in the group of communicators approves of the presence of all communicators.

Thus, group communication posits demands on the group of agents that communicates as well as on the group of agents that executes the action later on. The global variable  $g$  refers to both these groups. Alternatively, we could use two variables, one for the group of communicators and one for the group of actors. This would make the language more expressive, because different demands can be made on both groups. For example, in this variant of GrAPL it is possible to do action delegation, by establishing a constraint of the form  $g_a = \{\iota_1, \iota_2, \iota_3, \dots\}$  on the group of actors (denoted by  $g_a$ ). As this constraint doesn't apply to the group of communicators, these two groups can be different, while in the present formalisation of GrAPL the groups must be the same. In the next chapter, we present a coordination language which is similar in spirit to GrAPL. In this language we do make the distinction between negotiators and actors. There are advantages to using one variable to constrain the group of communicators and the group of actors. For example, when a constraint implies that agent Gareth has to be part of the group of agents executing the action `TakeTrip`, then Gareth also has to be present in the negotiation phase of this action. This way, we make sure that the autonomy of Gareth is respected. As Gareth can communicate his demands on the trip to be taken, he has no reason later on not to take part in the joint execution of action `TakeTrip`.

If an agent executes `CommGroupAdd`( $\varphi, a$ ), then it proposes its previously accumulated constraint on action  $a$  strengthened with  $\varphi$ . If an agent executes `CommGroupReset`( $\varphi, a$ ), then it erases its present constraint on  $a$  and offers  $\varphi$  as a fresh proposal. In both cases, the resulting constraint on  $a$  will be the conjunction of the proposals of all communicators. The local bindings of  $a$  are updated accordingly. If the agents disagree, the resulting constraint will be  $\perp$ . We allow this because  $\perp$  in a constraint store indicates that group communication has failed, and agents can test their constraint stores to find out whether this is the case. Subsequently, the resulting formula constrains each execution of  $a$  for each agent that has participated in the group communication, until the constraint on the action  $a$  is changed again. As the constraints are local and communication is synchronous, it is impossible for one agent to alter the constraints of another agent, without communicating with the other agent.

The syntax allows free local variables in `CommGroupAdd` statements and `CommGroupReset` statements. For these, we make the same restriction as we did for the free local variables in `ins` statements; they must be guarded, as a constraint containing a free local variable doesn't buy you much. When the free local variables in new constraints are guarded, they act as place-holders, lying in the scope of a binding statement (tests or action executions). This implies that at runtime, the local variables are instantiated with ground terms.

The last basic statement is action execution, denoted by  $\mathbf{a}(t_1, \dots, t_k)$  (sometimes abbreviated to  $\mathbf{a}(\bar{t})$ ). We use this statement both for individual action and group action. As explained earlier, global variables are forbidden in actual action parameters. The constraints associated with the action in the local states of agents trying to perform the action determine how many agents are needed, and sometimes make demands on their identities. If a group of agents (possibly consisting of only one member) tries to synchronously execute an action, the constraints of the agents on this action have to be consistent with each other and the actual parameters (the terms  $t_1, \dots, t_k$ ) and the group composition have to satisfy all constraints. In implementations of GrAPL, a constraint solver has to be plugged in to check this.

Another aspect of action execution is *execution time communication*. If one or more agents use a free local variable in an actual parameter, and at least one agent specifies a definite value for this parameter, then the last agent *communicates* the value to the other agent(s). Example 5.4 below illustrates this. This form of communication generates bindings to the free variables used by the listening agents.

The following example illustrates the novel statements.

#### EXAMPLE 5.4 (Jogging agents)

Two agents, James and Clare, arrange to go jogging. They discuss and subsequently execute  $\mathbf{Jog}(v_1, v_2)$ , where  $v_1$  and  $v_2$  are the formal parameters of the action. The first one is the time at which the agents start jogging, and the second parameter is the distance to be jogged.

Each agent has a constraint it wants to impose on the parameters of `Jog`. In these constraints, the agents use the formal parameters  $v_1$  and  $v_2$  to refer to the explicit parameters of the action. Each action also has one implicit parameter, denoted by  $g$ , which is the group composition. These are the constraints of Clare and James:

$$\begin{aligned} \text{James: } & \varphi : v_1 > 19.00 \wedge (v_2 = 7 \vee v_2 = 8) \wedge \text{Clare} \in g \\ \text{Clare: } & \psi : v_1 < 20.00 \wedge (v_2 = 8 \vee v_2 = 9) \wedge \\ & (v_1 > 19.00 \rightarrow \text{James} \in g) \end{aligned}$$

So, James wants to start jogging after 19.00 o'clock, he wants to run 7 or 8 km., and he wants Clare to join him. Clare on the other hand only wants James to jog with her when she leaves after 19.00 o'clock, she wants to start before 20.00 o'clock, and she wants to run 8 or 9 km.

They synchronously communicate:

James:  $\text{CommGroupReset}(\varphi, \text{Jog})$

Clare:  $\text{CommGroupReset}(\psi, \text{Jog})$

The result of this synchronous communication is a new constraint, which holds for future executions of  $\text{Jog}$  of both agents:

$$19.00 < v_1 < 20.00 \wedge v_2 = 8 \wedge \text{James} \in g \wedge \text{Clare} \in g$$

Next, the agents synchronously execute:

James:  $\text{Jog}(19.30, 8)$

Clare:  $\text{Jog}(x, 8)$

Note that there is execution-time communication here. James communicates the time 19.30 to Clare; Clare uses a free variable as first actual parameter, thereby indicating she is expecting James to pick the definite time. The constraint solver checks whether the actual parameters satisfy the constraints of James and Clare. This is the case, so the action is successful. In case James had performed  $\text{Jog}(y, 8)$  instead of  $\text{Jog}(19.30, 8)$ , there would have been multiple possibilities for the first parameter (namely, every time between 19.00 and 20.00). In this situation, one value is picked, or the overall action fails. When we define the semantics of GrAPL, we show several variants which differ in this aspect.

Having defined the set  $\mathcal{S}$  of basic statements, we now define the programs of GrAPL.

### DEFINITION 5.3 (Agent programs)

The set  $\mathcal{P}$  of valid single-agent programs is the smallest set containing the following programs:

- \*  $\alpha$ , where  $\alpha \in \mathcal{S}$ .
- \* if  $\varphi$  then  $\pi_1$  else  $\pi_2$ , where  $\varphi \in \mathcal{L}$ ,  $\text{var}(\varphi) \cap \mathcal{GV} = \emptyset$  and  $\pi_1, \pi_2 \in \mathcal{P}$ .
- \* if  $\varphi$  for a then  $\pi_1$  else  $\pi_2$ , where  $\varphi \in \mathcal{L}$ ,  $a \in \mathcal{A}$  and  $\pi_1, \pi_2 \in \mathcal{P}$ .
- \*  $\pi_1; \pi_2$ , where  $\pi_1, \pi_2 \in \mathcal{P}$ .
- \*  $\pi_1 + \pi_2$ , where  $\pi_1, \pi_2 \in \mathcal{P}$ .

We defined programs for single agents here. A multi-agent system simply is a set of single agent programs. These will be executed in parallel.

More complex programs can be formed using the if–then–else constructs, sequential composition and non-deterministic choice. The composed statement if  $\varphi$  then  $\pi_1$  else  $\pi_2$  first checks whether  $\varphi$  can be inferred from the belief base of the agent. If this is the case,  $\pi_1$  is executed, and if not,  $\pi_2$ . The statement if  $\varphi$  for a then  $\pi_1$  else  $\pi_2$  is similar, except that this statement tests the constraint bound to the action a. Inclusion of these statements is useful, because it enables testing whether something *can't* be inferred, which is not possible with the test statements  $?\varphi$  and  $?( \varphi, a )$ . Later on, in Examples 5.13 and 5.14, we will encounter if  $\perp$  for a then  $\pi_1$  else  $\pi_2$ . This checks whether the constraint on a has become inconsistent (because of group communication), and chooses an appropriate course of action. Statements like these allows the programmer to explicitly encode backtracking mechanisms in negotiation.

Repetitive behaviour is not essential for our approach. So, for technical convenience we didn't include iteration or recursion. Including iteration in the language would lead to problems. This is because we use variables in this language in a logical manner instead of in an imperative one. So, once a variable is bound to a value by a substitution, no other value can ever be bound to it. In an iterated loop, there typically are some variables that are assigned a new value in every round of the loop. This cannot be done in our language. A solution would be to rename variables when a new iteration starts, but this is not a very intuitive solution. In order to allow for (possibly infinite) repetitive behaviour, we could add recursive procedures to the language, in combination with variables local to program fragments. This is a well-studied technique. As we want to focus our attention at group coordination aspects, we will not pursue this. This matter is orthogonal to the issues discussed and the extension can be made in a straightforward manner.

## 5.4 Semantics

GrAPL has a formal operational semantics, in the style of Plotkin [98]. To define the semantics of an agent system, consisting of a number of agent programs in parallel, we first have to provide some definitions.

## 5.4.1 Preliminaries

First, we have to define the nature of agent configurations.

Each agent has a local configuration, which is a quadruple  $\langle \mu, \delta, \iota, \pi \rangle$ . The first element of the configuration is  $\mu$ , which stores the constraints bound to actions. These bindings are local to the agent. This way, it is possible for two agents to have different constraints for some action. This happens when the agents are part of separate groups which have communicated about the action independently. The constraints contain no free local variables. The function  $\mu$  is total, as each action is initially bound to  $\top$  (no constraints). When the constraint on a certain action  $\mathbf{a}$  is updated, we use the notation  $\mu[\psi/\mathbf{a}]$ , which denotes the function which is equal to  $\mu$  except for the constraint on  $\mathbf{a}$ , which has become  $\psi$ . The second configuration element is the belief base of the agent, denoted by  $\delta$ . We don't allow free variables in the belief base. Finally,  $\iota \in \mathcal{I}$  is the identity of the agent and  $\pi$  is the agent program fragment still to be executed.

DEFINITION 5.4 (Agent configuration)

A *local agent configuration*  $A_\iota$  is a quadruple  $\langle \mu, \delta, \iota, \pi \rangle$ . Here,

- \*  $\mu : \mathcal{A} \rightarrow \mathcal{L}$  where for all  $\mathbf{a} \in \mathcal{A} : \text{free}(\mu(\mathbf{a})) \cap \mathcal{L}\mathcal{V} = \emptyset$ .
- \*  $\delta \subseteq \mathcal{L}$ ,  $\text{free}(\delta) = \emptyset$ , and  $\text{var}(\delta) \cap \mathcal{G}\mathcal{V} = \emptyset$ .
- \*  $\iota \in \mathcal{I}$
- \*  $\pi \in \mathcal{P}$

A *multi-agent system* consists of a number of agents executing in parallel. A *global system configuration* simply is a set of local agent configurations of all agents present in the system. The set of agents present in a system is fixed. So, we just as well assume that  $\mathcal{I}$  is exactly the set of all agents in the system. Then, we have:

DEFINITION 5.5 (System configuration)

A *global system configuration* is a set  $\{A_\iota \mid \iota \in \mathcal{I}\}$  of local agent configurations.

As seen above, we use  $\mu$  to store the bindings to actions. But local variables can also be bound to values, for example when the belief base is tested. In our semantics we use *ground substitutions* to implement this. A substitution is ground if it binds variables to terms without variables in them.

DEFINITION 5.6 (Substitution)

- \* A *substitution*  $\theta$  is a finite set of pairs (also called *bindings*) of the form  $x_i := t_i$ , where  $x_i \in \mathcal{L}\mathcal{V}$  and  $t_i$  is a term of  $\mathcal{L}$ ,  $x_i \neq x_j$  for every  $i \neq j$ , and  $x_i \notin \text{free}(t_j)$  for every  $i$  and  $j$ .
- \* A *ground* substitution  $\theta$  is a substitution such that for every pair  $x := t \in \theta$  the term  $t$  is ground, i.e.  $\text{free}(t) = \emptyset$ .
- \* The *domain* of a substitution  $\theta$ , denoted by  $\text{dom}(\theta)$ , is the set of variables  $x$  for which  $\theta$  contains a pair  $x := t$ .

By definition, substitutions can only bind local variables. Global variables can be used as formal parameters for different actions and in many constraint stores. Creating a binding to a constraint variable thus could potentially influence the meaning of many constraint stores, so we don't allow it.

We define application of a substitution only informally, as it is a well-known notion and a complete formal definition would involve a lot of notational clutter.

DEFINITION 5.7 (Application of a substitution)

Let  $e$  be any syntactic expression, be it from  $\mathcal{L}$  or  $\mathcal{P}$ , and let  $\theta$  be a substitution. Then  $e\theta$  denotes the expression where all free variables  $x$  in  $e$  for which  $x := t \in \theta$  are simultaneously replaced by  $t$ .

We sometimes use substitutions to *unify* different sets of terms. We need some definitions for the notions of unifier and most general unifier.

DEFINITION 5.8 (Unifiers and most general unifiers)

- \* Let  $\theta$  and  $\eta$  be two substitutions. Then, the *composition*  $\theta\eta$  is the substitution  $\{(x := (x\theta)\eta) \mid x \in \text{dom}(\theta) \text{ or } x \in \text{dom}(\eta)\}$ .
- \* Let  $\{e_i \mid i = 1, \dots, n\}$  be a set of expressions. A substitution  $\theta$  is a *unifier* of these expressions if  $\forall i, j \in \{1, \dots, n\} : e_i\theta = e_j\theta$ .
- \* Let  $\{e_i \mid i = 1, \dots, n\}$  be a set of expressions. A substitution  $\theta$  is a *most general unifier* of these expressions if  $\theta$  is a unifier of  $\{e_i \mid i = 1, \dots, n\}$  and for all other unifiers  $\zeta$  it holds that  $\zeta = \theta\eta$ , for some substitution  $\eta$ .

So, the application of the composed substitution  $\theta\eta$  means that first  $\theta$  is applied, and then  $\eta$ . A unifier of a set of expressions is a substitution such that the expressions are all equal after application of the substitution, and a most general unifier is a unifier that keeps the expressions as general as possible.

The semantics of GrAPL we will give in the next two subsections is defined on two different levels, the *local agent level* and the *global system level*, like in the previous chapter. The reason we use a two-layered semantics is that the behaviour of a multi-agent system is an amalgamation of the interacting behaviours of the agents. Each agent can perform social actions (group communication and group action execution) and individual actions (the other statements). Individual actions don't depend on or influence the other agents. The meaning of these actions can be defined locally. The transitions of these actions are labelled with the symbol  $\tau$ . At the global system level, these actions simply are interleaved with the actions of other agents. The outcome of social actions depends on the behaviour of the other agents. The meaning of these actions can only be defined at the global system level. Nevertheless, a dummy local transition step is generated for social actions, labelled with the action details necessary to define the semantics of the social action globally.

The semantic model we use in this chapter is quite different from the model in the previous chapter. Here, we don't pay any attention to real-time aspects. We simply interleave the individual actions of the different agents, and let social actions shake hands by executing them synchronously. In the semantics of GrAPL, we abstract from interacting physical actions and from details of agent communication. Agents don't send messages back and forth in GrAPL; instead, group communication is synchronous. Thus, the two programming languages are on different abstraction levels. In devising the skeleton programming language of Chapter 4, our aim was to build a realistic, generic model of agent interaction, and here we want to design high-level communication primitives for group coordination. These different aims lead to different semantic models. It might be interesting to combine the two models into a new model, for example by adding real-time aspects to GrAPL, and allowing agents to state demands on the duration of group actions. We leave this for future work.

## 5.4.2 Local semantics

In general, a local transition looks like  $\langle \mu, \delta, \iota, \pi \rangle \xrightarrow{\tau}_\theta \langle \mu', \delta', \iota, \pi' \rangle$ . We use *labelled* transitions, because sometimes information present in the local level is needed to synchronously execute certain statements at the global level. The label  $\tau$  marks internal individual agent steps, resulting from local reasoning. When the statement locally executed is group communication or group action execution other labels are used to represent information necessary for the global semantics, as explained later on. The transition arrow is subscripted with a substitution, which contains bindings created by tests or communication that have to be passed on to the rest of the agent's program.

First, we give the transition rules for the basic statements. These are followed by the transition rules for composite programs. We again use the symbol  $\surd$  to denote the empty program remainder; this results if there are no more statements left to be executed.

The first transition rule we give is for doing nothing, that is, execution of skip.

$$\frac{}{\langle \mu, \delta, \iota, \text{skip} \rangle \xrightarrow{\tau} \langle \mu, \delta, \iota, \surd \rangle}$$

As expected, skip doesn't affect anything, except the program to be done next.

We continue with tests of the belief base and actions, respectively. Testing generally yields values for the free variables in the formula tested, that is, substitutions. For example, if the belief base contains the formula  $\text{Birthday}(\text{Wieke}, 12-12)$ , then performing the test  $?\text{Birthday}(\text{Wieke}, d)$  yields the value 12-12 for the variable  $d$ . The only difference between the two test statements is the domain of the substitution yielded. As there can occur global variables in formulas tested on constraints of actions, and bindings to global variables are undesirable, we exclude these from the domain of the substitution.

Let  $\theta$  be a ground substitution such that  $\text{dom}(\theta) = \text{free}(\varphi)$ .

Let  $\theta$  be a ground substitution such that  $\text{dom}(\theta) = \text{free}(\varphi) \setminus \mathcal{GV}$ .

$$\frac{\delta \vdash \varphi\theta}{\langle \mu, \delta, \iota, ?\varphi \rangle \xrightarrow{\tau} \langle \mu, \delta, \iota, \surd \rangle}$$

$$\frac{\mu(\mathbf{a}) \vdash \varphi\theta}{\langle \mu, \delta, \iota, ?(\varphi, \mathbf{a}) \rangle \xrightarrow{\tau} \langle \mu, \delta, \iota, \surd \rangle}$$

So, if tests are successful, they always yields ground terms for all free local variables in the tested formula. This doesn't have to mean that the belief base or constraint store uniquely determines a value for each variable. In case this is not so, there are multiple possible outcomes (substitutions yielded) for the test. The next example clarifies this.

#### EXAMPLE 5.5 (Tests in action)

We could have a belief base containing these three formulas:

$$\begin{aligned} \text{distance}(\text{Amsterdam}, \text{Utrecht}) &= 10 \\ \text{distance}(\text{Amsterdam}, \text{Rotterdam}) &= 13 \\ \text{distance}(\text{Amsterdam}, \text{Den Haag}) &= 10 \end{aligned}$$

If we test this belief base using the statement  $?distance(\text{Amsterdam}, x) = 10$  where  $x$  is a free variable, then there are two substitutions that can result, namely  $\{x := \text{Utrecht}\}$  and  $\{x := \text{Den Haag}\}$ . Similar things happen in tests of constraint stores. Look at this constraint on action  $a$ , which poses three demands on the first formal parameter of  $a$ :

$$\begin{aligned} distance(v_1, \text{Utrecht}) &= 10 \wedge \\ distance(v_1, \text{Rotterdam}) &= 13 \wedge \\ distance(v_1, \text{Den Haag}) &= 10 \end{aligned}$$

If we test this constraint store with  $?(distance(v_1, x) = 10, a)$  (meaning “For which city does the constraint on  $a$  imply that the distance to the first action parameter must be 10?”), then there are again two possible substitutions resulting, namely  $\{x := \text{Utrecht}\}$  and  $\{x := \text{Den Haag}\}$ .

In the transition resulting from the transition rule, the obtained values are stored in the substitution attached to the arrow. In the transition rules for the composite programs, these values will be substituted throughout the rest of the program. In case no suitable substitution can be found, there is no transition generated; the test fails.

Inserting something into the belief base causes a *belief revision*. This is by no means trivial; much research has been done on the subject (see eg. [49]). Here, we abstract from the belief revision process, by supposing a belief revision function  $\rho : \wp(\mathcal{L}) \times \mathcal{L} \rightarrow \wp(\wp(\mathcal{L}))$ . This function takes the old belief base and a formula to be inserted, and yields the set of new belief bases that could result from belief revision. One of these is non-deterministically chosen. Then, this is the transition rule for insertion into the belief base:

$$\frac{\delta' \in \rho(\delta, \varphi)}{\langle \mu, \delta, \iota, \text{ins}(\varphi) \rangle \xrightarrow{\tau} \emptyset \langle \mu, \delta', \iota, \sqrt{\phantom{x}} \rangle}$$

The condition of this transition rule formalises the process of belief revision. As insertion of a belief formula doesn’t yield any new information, the substitution yielded is the empty one. Note that we stipulated that free variables occurring in formulas inserted into the belief base are guarded. This means that when these insertions are executed, the free variables have been replaced by ground values from the domain  $\mathcal{D}$ . This is necessary as the belief base is defined to be a set of closed formulas.

Next are the `CommGroup`-statements, `CommGroupAdd` and `CommGroupReset`. Locally the resulting set of constraints cannot be found, as this is also

determined by the other communicators. Therefore, the update of bindings to actions takes place globally. The local transition is a dummy transition, which has the purpose to deliver information on the particulars of the CommGroup-statement, in the label of the transition arrow, to prepare a transition step of the whole system.

$$\frac{\langle \mu, \delta, \iota, \text{CommGroupReset}(\mathbf{a}, \varphi) \rangle \xrightarrow{\mathbf{a}::\varphi} \emptyset \langle \mu, \delta, \iota, \sqrt{\phantom{x}} \rangle}{\langle \mu, \delta, \iota, \text{CommGroupAdd}(\mathbf{a}, \varphi) \rangle \xrightarrow{\mathbf{a}::\mu(\mathbf{a}) \wedge \varphi} \emptyset \langle \mu, \delta, \iota, \sqrt{\phantom{x}} \rangle}$$

As these rules yield dummy transitions, the resulting configuration is not computed locally. So, the resulting configuration  $\langle \mu, \delta, \iota, \sqrt{\phantom{x}} \rangle$  at the right-hand sides of the transition arrows isn't the actual local configuration resulting from group communication. More specifically, in the rules above the constraint store component doesn't seem to be affected by the CommGroup, while we know that group communication updates the constraint stores. This update is computed globally, and overwrites the unchanged constraint store function  $\mu$ . The only relevant information in the resulting configurations is the program component, which will be built up by a bottom-up application of the local transition rules.

The label  $\mathbf{a} :: \psi$  above the transition reads "I propose to do  $\mathbf{a}$  under the constraint  $\psi$ ." In case of a CommGroupAdd-statement,  $\psi$  is the conjunction of the stored constraint on  $\mathbf{a}$  and  $\varphi$ , meaning that the agent strengthens its present constraint with  $\varphi$ . In case of a CommGroupReset-statement,  $\psi$  is  $\varphi$ , meaning that the agent overwrites its stored constraints and offers the fresh proposal  $\varphi$ .

Note that group communication doesn't yield any bindings to local variables; the resulting substitution is  $\emptyset$ . The only thing communicated are constraints on global variables; local variables are not involved in this form of communication.

Group action execution is influenced by all group members, so local semantics also yields a dummy transition labelled with action details, which serves to prepare a global transition step. Action execution can be done individually or groupwise. There is no real difference between these two options. Individual actions simply are group actions where the group only has one member. Locally, an agent tries to execute  $\mathbf{a}(\bar{t})$ . The terms  $\bar{t}$ , which are the actual parameters of the action  $\mathbf{a}$ , may contain free variables, indicating that the agent hasn't chosen a specific value for some formal parameters. In the global semantics, a substitution for these free variables is generated. Also globally, the local belief base is updated to reflect changes in information on the state of the world after the action has been done. Now, this is the local transition rule:

$$\frac{}{\langle \mu, \delta, \iota, \mathbf{a}(\bar{t}) \rangle \xrightarrow{\mathbf{a}(\bar{t})_{\emptyset}} \langle \mu, \delta, \iota, \sqrt{} \rangle}$$

We use the label  $\mathbf{a}(\bar{t})$ , which indicates that the agent intends to perform  $\mathbf{a}$  with actual parameters  $\bar{t}$ . Again, the fact that in the resulting configuration the belief base hasn't changed doesn't mean anything, as the change to the beliefs is computed globally.

Now, we arrive at the program constructors. We employ the convention that  $\sqrt{}; \pi$  equals  $\pi$ .

First, we define the semantics of the if-then-else-statements. These come in two variants, one which tests the belief base and one which tests the constraint on an action. The semantics of both variants is similar. There are two transition rules for each variant, one for a succeeding test and one for a failing test.

We start with if-then-else-statements that test the belief base. In the following transition rules, let  $\Theta$  be the set of ground substitutions  $\theta$  such that  $\text{dom}(\theta) = \text{free}(\varphi)$ .

$$\frac{\theta \in \Theta, \delta \vdash \varphi \theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\theta} \langle \mu, \delta, \iota, \pi_1 \rangle}$$

$$\frac{\nexists \theta \in \Theta : \delta \vdash \varphi \theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\emptyset} \langle \mu, \delta, \iota, \pi_2 \rangle}$$

Note that if the test fails, there is no substitution to propagate. These rules give the semantics of the test in the if-then-else statement, and the choice made between the two programs. It isn't surprising that these transition rules resemble those of  $?\varphi$  and  $?( \varphi, \mathbf{a} )$ .

For testing the constraint on  $\mathbf{a}$ , we have the following two rules, in which  $\Theta$  is the set of ground substitutions  $\theta$  such that  $\text{dom}(\theta) = \text{free}(\varphi) \setminus \mathcal{GV}$ .

$$\frac{\theta \in \Theta, \mu(\mathbf{a}) \vdash \varphi \theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ for } \mathbf{a} \text{ then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\theta} \langle \mu, \delta, \iota, \pi_1 \rangle}$$

$$\frac{\nexists \theta \in \Theta : \mu(\mathbf{a}) \vdash \varphi \theta}{\langle \mu, \delta, \iota, \text{if } \varphi \text{ for } \mathbf{a} \text{ then } \pi_1 \text{ else } \pi_2 \rangle \xrightarrow{\tau}_{\emptyset} \langle \mu, \delta, \iota, \pi_2 \rangle}$$

For sequential composition, the rule is entirely conventional, except for *substitution propagation*. A substitution resulting from testing or communicating must be applied to the remainder of the program left to be executed. If we

have  $\pi_1; \pi_2$  and executing the first statement of  $\pi_1$  yields a substitution  $\theta$  and a remaining program  $\pi'_1$ , then this substitution has to be applied to  $\pi_2$  when the sequential composition is executed.

$$\frac{\langle \mu, \delta, \iota, \pi_1 \rangle \xrightarrow{\iota} \theta \langle \mu', \delta', \iota, \pi'_1 \rangle}{\langle \mu, \delta, \iota, \pi_1; \pi_2 \rangle \xrightarrow{\iota} \theta \langle \mu', \delta', \iota, \pi'_1; \pi_2 \theta \rangle}$$

Note that the substitution isn't applied to  $\pi'_1$  in this rule. This is not necessary, because the transition in the antecedent of the rule takes care of this. The transition in the consequent of the rule still carries the substitution  $\theta$ , as there might be other parts of the program not mentioned in the rule, to which the substitution still has to be applied.

To clarify why  $\theta$  isn't applied to  $\pi'_1$ , we use the following example.

**EXAMPLE 5.6** (Substitution propagation)

Look at this simple program:

(?Birthday(Wieke,  $d$ ); buy\_present\_before( $d$ )); give\_present(Wieke,  $d$ )

To compute the semantics of this program, we start with the semantics of the first atomic statement. For ease of explanation, we leave out all elements of the local agent configuration, except the program. Now, the transition rule for testing the belief base yield this transition:

$$?\text{Birthday}(\text{Wieke}, d) \xrightarrow{\tau}_{d:=12-12} \checkmark$$

Because of the above rule for sequential composition, we then have

$$\begin{aligned} ?\text{Birthday}(\text{Wieke}, d); \text{buy\_present\_before}(d) &\xrightarrow{\tau}_{d:=12-12} \\ &\checkmark; \text{buy\_present\_before}(12-12) \end{aligned}$$

The resulting program is equivalent to  $\text{buy\_present\_before}(12-12)$ . Then the first transition step of the entire program becomes:

$$\begin{aligned} &(\text{?Birthday}(\text{Wieke}, d); \text{buy\_present\_before}(d)); \text{give\_present}(\text{Wieke}, d) \\ &\xrightarrow{\tau}_{d:=12-12} \\ &\text{buy\_present\_before}(12-12); \text{give\_present}(\text{Wieke}, 12-12) \end{aligned}$$

Note that the substitution  $d := 12-12$  already has been applied to the statement  $\text{buy\_present\_before}(d)$  in the computation of the previous transition.

Next are the rules for non-deterministic choice.

$$\frac{\langle \mu, \delta, \iota, \pi_1 \rangle \xrightarrow{l} \langle \mu', \delta', \iota, \pi'_1 \rangle}{\langle \mu, \delta, \iota, \pi_1 + \pi_2 \rangle \xrightarrow{l} \langle \mu', \delta', \iota, \pi'_1 \rangle}$$

$$\frac{\langle \mu, \delta, \iota, \pi_2 \rangle \xrightarrow{l} \langle \mu', \delta', \iota, \pi'_2 \rangle}{(\sigma, \delta, \iota, \pi_1 + \pi_2) \xrightarrow{l} (\sigma', \delta', \iota, \pi'_2)}$$

Again, the substitution has already been processed on the non-deterministic alternative chosen, and it only needs to be propagated.

### 5.4.3 Global semantics

To obtain the semantics of a multi-agent program, we use an interleaving semantics with a handshaking mechanism for synchronisation. Group communication and group action need to shake hands; the other basic statements are interleaved.

We start with group communication. When a group communicates about an action, the local constraints of the agents communicating are updated to be the conjunction of the proposed constraints. This might be an inconsistent formula, if the agents have conflicting interests. The resulting constraints can *control* the participation of agents in the *future execution* of the action communicated about, but we also want to *control the group of agents that takes part in the group communication itself*. If, for example, one or more agents demand that agent Jane should be excluded from the *group doing* the action discussed, it could be useful to forbid this agent to be part of the *group negotiating* the constraints on the execution of the action. So, we demand that agents may only participate in communication about the details of a future group action as long as presence of these agents is not inconsistent with the demands of each agent on the group composition. If the group communicating violates the demands of (at least) one group member, then the group communication fails, in the sense that no global transition is generated. Of course, the demands on the group composition must also hold when the action is to be executed by some group.

We need only one transition rule to cater for both `CommGroupAdd` and `CommGroupReset` statements, as the only difference between them is the proposed constraint (either a strengthening of the current constraint or a completely fresh proposal). We dealt with this locally; the label of the local transition contains the constraint proposed. Now, this is the transition rule for group communication:

Let  $J \subseteq \mathcal{I}$  be a set of agent names and let  $A_\iota = \langle \mu_\iota, \delta_\iota, \iota, \pi_\iota \rangle$ .

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\mathbf{a}::\psi_\iota} \emptyset \langle \mu_\iota, \delta_\iota, \iota, \pi'_\iota \rangle}{\{A_\iota \mid \iota \in \mathcal{I}\} \longrightarrow \{\langle \mu'_\iota, \delta_\iota, \iota, \pi'_\iota \rangle \mid \iota \in J\} \cup \{A_\iota \mid \iota \in \mathcal{I} \setminus J\}}$$

where  $\mu'_\iota = \mu_\iota[\bigwedge_{\iota \in J} \psi_\iota / \mathbf{a}]$  and the following condition holds:

$$\text{for each } \iota \in J \text{ it holds that } g = J \wedge \psi_\iota \not\vdash \perp$$

The antecedent of this transition rule consists of (dummy) local transitions for group communication. The labels above the arrows have to match on  $\mathbf{a}$ , the action discussed. Each agent  $\iota$  brings its own constraint,  $\psi_\iota$ . As a result,  $\mathbf{a}$  is constrained by  $\bigwedge_{\iota \in J} \psi_\iota$  in the new constraint stores  $\mu'_\iota$ . Note that this globally computed update of the constraint stores overwrites the resulting constraint stores from the dummy local transitions (which are still  $\mu_\iota$ , like in the configurations before execution of the group communication). The group communication takes place in a synchronised execution step. In this respect, `CommGroup` execution is similar to the synchronous communication primitives in (for example) CSP [61]. For the agents not participating in the communicative action, the local state stays the same.

The condition that for each  $\iota \in J$  it holds that  $g = J \wedge \psi_\iota \not\vdash \perp$  controls the composition of the group of agents communicating about the action. The formula  $g = J$ , stating that  $g$  is the group of communicating agents, should be consistent with the constraints of the agents. If the constraint of one of the communicating agents implies that there have to be at least three agents involved in the action, then each group communicating about this action must also contain at least three agents. As another example, if the constraint of one of the agents (say  $\iota_1$ ) implies that agents  $\iota_3$  and  $\iota_6$  have to participate in the group action, then group communication with  $\iota_1$  in the group can't succeed if these agents don't put in a word.

Note that the transition rule and the associated condition *don't* demand that the constraints of the communicating agents are consistent, that is, that  $\perp$  can't be inferred from  $\bigwedge_{\iota \in J} \psi_\iota$ . As long as the group of communicators is consistent with the constraints of the agents, the group communication succeeds. In case the agents have conflicting demands on action details, the constraint on  $\mathbf{a}$  will become  $\perp$  in the constraint stores of the agents. The reason for this choice, which may seem strange, is that  $\perp$  in a constraint store has a signalling function. The agent can test its constraint store, and when it turns out that the constraint is  $\perp$ , the agent knows that there was disagreement in an earlier negotiation round. Subsequently, the agent can try to re-initiate the communication about the group action by performing a `CommGroupReset`, which will remove

$\perp$  from the constraint store if the constraints of the agents communicating in this new negotiation round aren't inconsistent.

It is important to note that group communication essentially is non-deterministic. The resulting constraint store is fixed, being the conjunction of the constraints of the communicators, so this is not the non-deterministic element. But the group of agents communicating allows many possibilities, as long as the constraints are not too strong, as we show in the following example.

**EXAMPLE 5.7** (Non-determinism in group communication)

Some agent called Peter wants to form a group of agents that will help him in painting his new house. He needs at most four people to help him, and he would like Eunice to help, as she is very good at painting. Then, he could come up with a constraint  $\varphi$  which is the formula  $Eunice \in g \wedge Peter \in g \wedge \#(g) \leq 5$ . What could happen if Peter performs  $CommGroupReset(\varphi, PaintHouse)$ ? This generates a dummy local transition labelled with  $PaintHouse :: \varphi$ . One scenario is that indeed Eunice and exactly three other agents are prepared to answer with a synchronous  $CommGroup$ -statement. In this case, it is not at all certain that all four of these agents will synchronously communicate with Peter. The semantics defined above allows any group of agents that is consistent with the demands of the participants. So, if the four agents have no additional demands, the only agent who will certainly communicate is Eunice, as Peter has demanded her to be present. Any subset of the other three agents could also communicate with Peter and Eunice; for now, the choice is a non-deterministic one.

But it could also happen that the only agent reacting to Peter's request is Eunice. The group communication will succeed, as this group also satisfies the demands on the group. Another situation occurs when twenty agents state their willingness to assist Peter. Then, each group with at least Eunice and Peter in it, and at most five members, could end up communicating about the painting of Peter's house, as well as groups without Peter. We go into this last option. If Peter isn't part of the communicating group, then his demands don't need to be satisfied, and other communicators can come to very different agreements on the painting of the house.

It is possible to limit this kind of non-determinism by adding another demand to the transition rule for group communication. We could demand the communicating group to be maximal or minimal with respect to set inclusion. But we won't make a decision on this issue, as this choice is influenced by the specific domain of the multi-agent system.

The transition rule for action execution poses a dilemma. We have to decide in which way actual parameters of an action can be determined when the agents executing the action are not specific about them. To make matters clear, here's an example.

EXAMPLE 5.8 (Jogging again)

We revisit agents James and Clare, that have negotiated about jogging in the park together. As before, the action Jog has two explicit parameters, the first being the starting time and the second one the distance to be jogged. James and Clare have agreed upon constraints on these parameters, so their constraint stores associated with Jog are the same. They contain the constraint:  $g = \{\text{James, Clare}\} \wedge v_1 = 19.00 \wedge 8 \leq v_2 \leq 10$ . These constraints are not decisive; there are still three possibilities for  $v_2$  (assuming the value is a natural number). So, what happens when the agents try to jog together? There are a number of cases; we will give a representative example of each kind:

1. James and Clare both try to execute  $\text{Jog}(19.00, 8)$ . So, their action parameters agree and satisfy the constraints. The action will take place.
2. James tries  $\text{Jog}(x_1, 8)$  and Clare tries  $\text{Jog}(19.00, y_2)$ , where both  $x_1$  and  $y_2$  are free (local) variables. This implements execution time communication: James tells Clare that they will run 8 km., and Clare communicates that they will start at 19.00. All parameters are determined and satisfy the constraints, so the action  $\text{Jog}(19.00, 8)$  will be jointly executed.
3. James tries  $\text{Jog}(x_1, 8)$  and Clare doesn't feel like making any decisions, so she tries  $\text{Jog}(y_1, y_2)$ . The outcome of this is not immediately clear. It is clear that James communicates the distance parameter to Clare. The first parameter is not talked about, and seen mathematically, this is not necessary either, as the value of  $v_1$  is fixed by the constraints agreed upon. So, one choice is to let the above group action succeed;  $\text{Jog}(19.00, 8)$  is synchronously executed. The other choice is to let this group action fail, because there is no run-time agreement on all action parameters. There is something to say for both options.
4. James tries  $\text{Jog}(19.00, x_2)$  and Clare tries  $\text{Jog}(19.00, y_2)$ . It seems intuitively justified that this group action has to fail, as there is no clarity about the distance to be jogged, neither in the action parameters of the agents nor in the constraints. But another view is that

the value space for the second parameter still contains three possible values, and therefore there are three possible executions of the above statement, in which James and Clare jog 8, 9 or 10 kilometres, respectively.

As suggested by the example, it is not immediately clear what is *the* right semantics for action execution. In this chapter, we create three alternative semantics. One of these yields success for cases 3 and 4 above, the second failure for both and the third yields success for case 3 and failure for case 4. Before presenting these options, we will pinpoint the subtle semantical issues at hand.

In group formation and group action, there are two types of communication. There is communication *during the negotiation phase*, performed through one or more CommGroup-statements. Also, there is *execution time communication*, which takes place if some agents participating in the actual group action don't instantiate all action parameters with ground values, but use free local variables for these. Typically, *during the negotiation phase* each agent will make sure all its important demands on the action are incorporated into the set of constraints agreed upon. An agent might be willing to drop some of its less important demands, if this is the only way to form a group, but it will usually hold on to its major constraints. After group formation, the constraint stores of the agents need not be decisive on each action parameter. Often, there will still be a *parameter space* from which each choice is perfectly acceptable to all agents involved. The choice from this parameter space can be made *during the execution phase*. An agent can pick a value for an action parameter from the set of values allowed by the constraint on the action, and use this value for the parameter. This is what happened in case 2 of the example: the distance wasn't fixed by the constraints yet, so James chooses the distance to be 8 by trying to execute  $\text{Jog}(x_1, 8)$ . Clare tries  $\text{Jog}(19.00, y_2)$ , and because she didn't pick a value for the distance parameter, she lets James determine the distance.

The combination of execution time communication and negotiation phase communication makes matters opaque here. For ease of formulation, we introduce a new term. We call a formal parameter of an action *a definite* for a group of agents  $J$  if the conjunction of the constraints associated with  $a$  by the agents in  $J$  allows only one value for that parameter. This is the formal definition:

**DEFINITION 5.9** (Definite action parameters)

Let  $J \subseteq \mathcal{I}$  be a set of agent identities, and let  $\langle \mu_\iota, \delta_\iota, \iota, \pi_\iota \rangle$  be the local configuration of agent  $\iota$ . A formal parameter  $v_k$  of an action  $a$  is *definite* for  $J$  if there exists a value  $d \in \mathcal{D}$ , such that  $\bigwedge_{\iota \in J} \mu_\iota(\mathbf{a}) \not\vdash \perp$  and  $\bigwedge_{\iota \in J} \mu_\iota(\mathbf{a}) \vdash v_k = d$ .

We will sometimes be sloppy, and call action parameters definite without referring to an agent group. In the above definition,  $\bigwedge_{i \in J} \mu_i(\mathbf{a})$  is the joint constraint of the agents in  $J$  on action  $\mathbf{a}$ . If this constraint implies that the formal parameter  $v_k$  must have a certain ground value  $d$ , then this parameter is definite. We need to exclude the case that the joint constraint is inconsistent, as then everything can be derived from it, and the constraint thus determines nothing.

In case 3 of the example above, the essential action parameter is the first one, the time James and Clare will go jogging. This parameter *is definite* for the agents; their constraint stores fix the time at 19.00. So, group communication about the constraints on Jog has settled on a value. Now we focus solely on execution time communication. During execution time communication, both agents use a free variable for the first parameter, thereby indicating that they are willing to let the other agent determine the value of the first parameter. But as none of the agents supplies a value, execution time communication alone can't fix the first parameter. Still focusing only on execution time communication, we can choose between two possible scenarios: the communication could succeed where the value is non-deterministically chosen from the domain of the first parameter, or the communication, and thereby the overall group action, could fail. In the former case, the constraints of both agents will single out the only possible group action parameter.

In case 4, the second parameter is the essential one. This parameter *is not definite*; the constraints still allow three possible values. But actually this is the only difference between case 3 and case 4 of the example. Both agents again use a free variable for the parameter in their action call, so again run-time communication could fail or yield success with a non-deterministic outcome. In case we choose the second option, the range of possible values for the second parameter will be considerably narrowed by the constraint stores, yielding three possible action executions.

So, from these considerations it seems that there are two options for the semantics of group action. In the first, execution time communication doesn't have to be conclusive to yield action success. If it isn't, then actual action parameters are chosen in a non-deterministic manner. In the second option, we demand run-time communication to be conclusive, and the action will fail if this isn't the case. There is still a third option, which lies in between these two alternatives.

Although there are three options for the semantics, we don't need three different transition rules. The difference will be made in the subtle choices in the conditions associated with the rule. First, we will present the transition rule with the most permissive conditions, formalising the semantics where run-time communication need not be conclusive.

Group action execution has a synchronised semantics, just like group communication. Action execution can be done individually or group-wise. Individual actions simply are group actions where the group has only one member. There are two conditions associated with action execution. First, the actual action parameters of all agents in the group have to be compatible. We model this by requiring the parameters to be *unifiable*. So, if for example one agent in the group tries the value 5 for the first action parameter, and another group member tries 6, then no successful group action is possible. Secondly, the constraints of all agents in the group on the action have to be obeyed. This is the *monitoring* of the action execution; the group action has to be executed according to the agreements made by the agents prior to the action.

Action execution might change the belief base of the agent. We assume that for each agent  $\iota$  we have a belief revision function  $\xi_\iota$ , which models this change. This function takes an action with definite parameters and the old belief base, and returns the set of belief bases that could result from a revision of the belief base with the effects of the action. One of these candidates is non-deterministically chosen. By introducing the function  $\xi_\iota$ , we abstract from details of belief revision; again, see [49] for details.

For the global transition for action synchronisation, we use the same conventions as in the previous transition rule. So, let  $J \subseteq \mathcal{I}$  be a set of agent names, let  $A_\iota = \langle \mu_\iota, \delta_\iota, \iota, \pi'_\iota \rangle$ , let  $\theta$  be a ground substitution and let  $a$  be an action with formal parameters  $\bar{v}$ .

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{a(\bar{t}_\iota)}_\emptyset \langle \mu_\iota, \delta_\iota, \iota, \pi'_\iota \rangle}{\{A_\iota \mid \iota \in \mathcal{I}\} \longrightarrow \{\langle \mu_\iota, \delta'_\iota, \iota, \pi'_\iota \theta \rangle \mid \iota \in J\} \cup \{A_\iota \mid \iota \in \mathcal{I} \setminus J\}}$$

where  $\delta'_\iota \in \xi_\iota(a(\bar{t}_\iota \theta), \delta_\iota)$  and the following conditions hold:

- \*  $dom(\theta) = \bigcup_{\iota \in J} free(\bar{t}_\iota)$  and  $\theta$  is a unifier of  $\{\bar{t}_\iota \mid \iota \in J\}$
- \*  $\bigwedge_{\iota \in J} \mu_\iota(a) \wedge \bar{v} = \bar{c} \wedge g = J \not\vdash \perp$ , where  $\bar{c} = \bar{t}_\iota \theta$  for any  $\iota \in J$

The first condition above demands that the substitution  $\theta$  provides a ground value for the free variables in the actual action parameters of all agents. Moreover, it states that  $\theta$  has to be a *unifier* of the actual parameter settings of all agents in the group; this means that  $\bar{t}_\iota \theta = \bar{t}_\kappa \theta$  for each  $\iota, \kappa \in J$ . This explains why the definition of  $\bar{c}$  (a shorthand for the unified actual parameters) in the second condition ends with ‘for any  $\iota \in J$ ’. The substitution  $\theta$  instantiates all free variables in the action parameters of the agents in the group in such a way, that all agents use the same set of actual parameters. Only this way, the action can be done in a coordinated manner.

The action with the unified actual parameters constitutes an input to the belief update function. Also, the substitution  $\theta$  is applied to the programs left to be executed, because in general action execution generates bindings to free variables which have to be passed on to the remainders of the programs. Both these updates overwrite elements in the resulting configurations of the dummy local transitions.

The second condition states that the actual values of the parameters of the group action and the composition of the group have to be consistent with the constraints of all participating agents. We explain this formula in detail. The formula  $\bigwedge_{i \in J} \mu_i(\mathbf{a})$  is the result of adding up the constraints that the participating agents associated with  $\mathbf{a}$ . To this formula, two conjuncts are added. The first formula,  $\bar{v} = \bar{c}$ , states the unified actual values of the formal parameters. The second formula,  $g = J$ , states the actual composition of the group about to perform the action. If there is a clash between the constraints of the agents and the actual action parameters, then an inconsistency can be derived, resulting in failure of the execution.

In an implementation of the language, a constraint solver can check consistency. Sometimes, a group action execution allows more than one possible unifier  $\theta$ . In this semantic variant, we leave this non-determinism unresolved; alternatively, it could give rise to failure, as is the case in the semantic variant we will present later on.

Looking back at the example of James and Clare going jogging, this choice of conditions implements the case that both case 3 and case 4 succeed:

#### EXAMPLE 5.9

The constraint on Jog is

$$g = \{\text{James, Clare}\} \wedge v_1 = 19.00 \wedge 8 \leq v_2 \leq 10.$$

We revisit the third and fourth case of Example 5.8:

3. James tries Jog( $x_1, 8$ ) and Clare tries Jog( $y_1, y_2$ ). As the free variables in the actual parameters are  $\{x_1, y_1, y_2\}$ , this is the domain of  $\theta$ . The substitution  $\theta$  must be a ground substitution, and a unifier of  $\{(x_1, 8), (y_1, y_2)\}$ . It is clear that  $y_2\theta$  must be 8. Furthermore,  $x_1\theta = y_1\theta = t$ , where  $t$  can be any time point according to the first condition of the global transition rule. The second condition demands that the unified actual parameters obey the constraint on Jog, which means that the only legitimate value for  $t$  is 19.00. The instantiated action Jog(19.00, 8) is executed.

4. James tries  $\text{Jog}(19.00, x_2)$  and Clare tries  $\text{Jog}(19.00, y_2)$ . Now,  $\text{dom}(\theta) = \{x_2, y_2\}$ . The ground substitution  $\theta$  must be a unifier of  $\{(19.00, x_2), (19.00, y_2)\}$ . In order to unify  $x_2$  and  $y_2$ , any distance can be assigned to them, but as the constraint on the distance must be obeyed there are three options:  $x_2\theta = y_2\theta = 8$ ,  $x_2\theta = y_2\theta = 9$  and  $x_2\theta = y_2\theta = 10$ . One of these options is randomly picked, and the Jog action takes place accordingly.

So, with these conditions, run-time communication need not be conclusive. This happens if all agents in the group  $J$  use a variable for a particular parameter of  $\mathbf{a}$ . The substitution  $\theta$ , being ground, fixes a value for this parameter in an arbitrary manner. The second condition of the transition rule checks this value against the aggregate constraints of the agents. If the parameter is definite (that is, completely determined by the constraints), then the range of values that  $\theta$  can assign to the parameter is reduced to one possibility. (This matches case 3 of the jogging example.) If the parameter is not definite, then multiple substitutions are possible, yielding a non-deterministic action execution. (This matches case 4 of the example.)

If we want to implement the option for the semantics where execution time communication has to be conclusive in order to successfully execute the group action, we have to replace the first demand of the global transition rule for action synchronisation, which was:

$$* \text{dom}(\theta) = \bigcup_{\iota \in J} \text{free}(\bar{t}_\iota) \text{ and } \theta \text{ is a unifier of } \{\bar{t}_\iota \mid \iota \in J\}$$

with this one:

$$* \text{dom}(\theta) = \bigcup_{\iota \in J} \text{free}(\bar{t}_\iota) \text{ and } \theta \text{ is the most general unifier of } \{\bar{t}_\iota \mid \iota \in J\}$$

With this demand, there is only one permissible ground substitution  $\theta$ , namely the most general unifier of the actual action parameters of all agents. In case there are one or more action parameters for which none of the agents determines a domain value, then the most general unifier will not be a ground substitution, and the group action can't be successful. Only if a value for each action parameter is fixed solely through run-time communication (without looking at the constraints), this variant of the semantics will yield action success. So, both the third and the fourth case of Example 5.8 above will fail with this semantic variant:

### EXAMPLE 5.10

We look at the third and fourth case again:

3. James and Clare try to execute  $\text{Jog}(x_1, 8)$  and  $\text{Jog}(y_1, y_2)$ , respectively. The most general unifier of these action parameters is  $\{x_1 := w, y_1 := w, y_2 := 8\}$ , where  $w$  is some variable, which isn't a ground substitution. So, this group action won't generate a global transition step.
4. James and Clare attempt  $\text{Jog}(19.00, x_2)$  and  $\text{Jog}(19.00, y_2)$ , respectively. The most general unifier of the parameter tuples is  $\{x_2 := u, y_2 := u\}$ , where  $u$  is a variable. Again, this isn't a ground substitution, so no global transition results.

There is a third option for the semantics. In this variant, case 3 of the example succeeds and case 4 fails. More generally speaking, group action execution is successful if each formal action parameter is definite for the group attempting the action or run-time communication is conclusive about its actual value. If this is so, there is only one unifying substitution that yields actual parameters satisfying the constraints of the agents. For this semantical variant, we add an extra condition to the two original conditions of the global transition rule for action synchronisation, resulting in these conditions:

- \*  $\text{dom}(\theta) = \bigcup_{\iota \in J} \text{free}(\bar{t}_\iota)$  and  $\theta$  is a unifier of  $\{\bar{t}_\iota \mid \iota \in J\}$
- \*  $\bigwedge_{\iota \in J} \mu_\iota(\mathbf{a}) \wedge \bar{v} = \bar{c} \wedge g = J \not\vdash \perp$ , where  $\bar{c} = \bar{t}_\iota \theta$  for any  $\iota \in J$
- \* There is only one ground substitution  $\theta$  that satisfies the two conditions above.

We again revisit the jogging example:

### EXAMPLE 5.11

The third and fourth case in this semantic variant:

3. James and Clare try to execute  $\text{Jog}(x_1, 8)$  and  $\text{Jog}(y_1, y_2)$ , respectively. In Example 5.9, we have seen that there is only one unifying ground substitution, such that the constraints of James and Clare are respected. This means that the group action Jog will successfully be executed, with actual parameters 19.00 and 8.
4. James and Clare attempt  $\text{Jog}(19.00, x_2)$  and  $\text{Jog}(19.00, y_2)$ , respectively. In Example 5.9, there were three candidate unifying ground substitutions, such that the constraints of the agents are obeyed by the instantiated actual parameters. So, the Jog action fails.

It might very well be possible to devise more, also interesting, variants of the global semantics for group action.

Group communication and group action execution are the only statements that are synchronously executed by a group of agents. All other statements of individual agents can simply be interleaved. All local transitions of these statements are labelled with the symbol  $\tau$ . So, let  $\iota \in \mathcal{I}$  be some agent taking a local execution step, let  $A_{\kappa} = \langle \mu_{\kappa}, \delta_{\kappa}, \kappa, \pi_{\kappa} \rangle$  and  $A'_{\kappa} = \langle \mu'_{\kappa}, \delta'_{\kappa}, \kappa, \pi'_{\kappa} \rangle$ , where  $\kappa$  is some element of  $\mathcal{I}$ .

$$\frac{A_{\iota} \xrightarrow{\tau}_{\theta} A'_{\iota}}{\{A_{\kappa} \mid \kappa \in \mathcal{I}\} \longrightarrow \{A'_{\iota}\} \cup \{A_{\kappa} \mid \kappa \in \mathcal{I} \setminus \{\iota\}\}}$$

One agent program is executed for one transition step, while the configurations of the other agents don't change.

## 5.5 Illustrations

To illustrate the mechanisms of the semantics and show the usefulness of our language, we will give three additional examples. The first example is very simple in nature, and we use it to explain the functioning of the two-layered semantics.

### EXAMPLE 5.12 (Arranging a meeting)

Dr. Van der Broek and Prof. Meyer are two scientists who would like to cooperate with each other. So, they want to make an appointment for a meeting which lasts a day. They communicate with each other about the action Meet. This action has one explicit parameter, the date of the meeting.

These are the programs they employ:

Program of Van der Broek:

```
CommGroupReset( $g = \{\text{Van der Broek, Meyer}\} \wedge$ 
                ( $v_1 = 26-1 \vee v_1 = 29-1 \vee v_1 = 30-1$ ), Meet);
if  $v_1 = x$  for Meet
then Meet( $x$ )
else CommGroupAdd( $v_1 = 26-1 \vee v_1 = 29-1$ , Meet); Meet( $y$ )
```

Program of Meyer:

```
CommGroupAdd( $g = \{\text{Van der Broek, Meyer}\} \wedge$ 
              ( $v_1 = 29-1 \vee v_1 = 30-1 \vee v_1 = 31-1$ ), Meet);
if  $v_1 = w$  for Meet
then Meet( $w$ )
else CommGroupAdd( $\top$ , Meet); Meet( $z$ )
```

Informally, these programs mean the following. Both Van der Broek and Meyer have the demand that the group which meets has to consist of the two of them. Also, they both initially propose three possible dates for the meeting. Note that two of these dates are possible to both scientists. They communicate with each other, which succeeds because the group communicating  $\{\text{Van der Broek, Meyer}\}$  satisfies the constraints of both agents. If another agent would have tried to join in the negotiation, this would not have succeeded. Van der Broek uses a `CommGroupReset` statement and Meyer uses a `CommGroupAdd` statement in the first group communication. As we assume that the constraint on `Meet` for both agents is  $\top$  before execution of the multi-agent program, it doesn't matter which of both `CommGroup` statements the agents use in the first communication round, as the resulting demand proposed turns out the same ( $\varphi = \top \wedge \varphi$ ). The updated constraint stores of both agents then contain the formula  $g = \{\text{Van der Broek, Meyer}\} \wedge (v_1 = 29-1 \vee v_1 = 30-1)$  as the new constraint on `Meet`. Then, both agents test whether their constraints fix one date for the action parameter, using the test in the if-then-else statement (we will explain the working of this test later on). If this would be so, they would meet at this date. But in this case, there are still two dates possible. So, Meyer leaves it up to Van der Broek to strengthen her preferences, as he adds the empty constraint  $\top$  to his present constraint. Van der Broek then narrows down her possible dates to two, and the scientists communicate for the second time. Now, their constraint stores allow only one date, which is 29-1. So, they meet then.

We now look at how the interplay of local and global semantics yields a trace of this system. Each global trace represents an actual computation of the system. Local transitions play an auxiliary role in the construction of global traces. To build a system trace, we alternately apply local and global transition rules. Local transitions of individual actions are interleaved in the global trace, and dummy local transitions for group actions lead to one synchronised global transition.

We start with the local semantics of the first `CommGroup` statements in both programs. The local transition rule for `CommGroupReset` yields a

dummy transition for Van der Broek, with labelled arrow  $\xrightarrow{\text{Meet}::\varphi_1} \emptyset$ , where  $\varphi_1 \equiv g = \{\text{Van der Broek, Meyer}\} \wedge (v_1 = 26-1 \vee v_1 = 29-1 \vee v_1 = 30-1)$ . Similarly, the local transition rule for `CommGroupAdd` yields a dummy local transition for Meyer, with labelled arrow  $\xrightarrow{\text{Meet}::\psi_1} \emptyset$ , where  $\psi_1 \equiv g = \{\text{Van der Broek, Meyer}\} \wedge (v_1 = 29-1 \vee v_1 = 30-1 \vee v_1 = 31-1)$ . The global transition rule for group communication checks whether the group of communicators satisfies the demands of both agents, and as this is the case, updates the constraints of both agents on `Meet` to be the conjunction of the proposals, which is (equivalent to)  $g = \{\text{Van der Broek, Meyer}\} \wedge (v_1 = 29-1 \vee v_1 = 30-1)$ .

Now, both agents locally test their constraint on `Meet`, as part of the if-then-else statements. We focus at the test of Van der Broek; the test of Meyer is analogous. The formula tested is  $v_1 = x$ , where  $x$  is a free local variable. We look at the local transition rule for if-then-else statements that test the constraint stores. If  $\mu$  is the present constraint store function of Van der Broek, then a ground substitution  $\theta$  is sought with  $\text{dom}(\theta) = \text{free}(v_1 = x) \setminus \mathcal{GV} = \{v_1, x\} \setminus \mathcal{GV} = \{x\}$ , such that  $(v_1 = x)\theta$  follows from the constraint  $\mu(\text{Meet})$ . As the current constraint on `Meet` still allows two values for  $v_1$  ( $v_1 = 29-1 \vee v_1 = 30-1$ ), such a substitution can't be found. The test for definiteness thus fails, and the second transition rule for the if-then-else statement is used, which results in Van der Broek taking the else branch. In case the constraint would have been  $v_1 = 29-1$ , then the test would have succeeded, yielding the substitution  $\{x := 29-1\}$ . This substitution then would have been applied to the rest of the program of Van der Broek, resulting next in an execution of `Meet(29-1)`. Like Van der Broek, Meyer also tests the constraint on `Meet` and takes the else branch. As these tests are individual actions, the global semantics interleaves them.

Both agents again arrive at a group communication statement, a `CommGroupAdd` in both cases. Meyer chooses not to strengthen the current constraint on `Meet`; he adds  $\top$ , resulting in a transition  $\xrightarrow{\text{Meet}::\psi_2} \emptyset$ , where  $\psi_2 \equiv g = \{\text{Van der Broek, Meyer}\} \wedge (v_1 = 29-1 \vee v_1 = 30-1)$ . Van der Broek adds the constraint  $v_1 = 26-1 \vee v_1 = 29-1$ , resulting in a dummy local transition with arrow  $\xrightarrow{\text{Meet}::\varphi_2} \emptyset$ , where  $\varphi_2 \equiv g = \{\text{Van der Broek, Meyer}\} \wedge v_1 = 29-1$ . The global transition rule combines these dummy local steps into a synchronous global step, in which the constraint on `Meet` of both agents is updated to be  $\varphi_2$ . Now, the date has been agreed upon.

Finally, both agents try a Meet-action, where the actual parameter of Van der Broek is the free local variable  $y$  and the actual parameter of Meyer is the free local variable  $z$ . The local transition rule for action execution generates two dummy local transition, with arrows  $\xrightarrow{\text{Meet}(y)} \emptyset$  and  $\xrightarrow{\text{Meet}(z)} \emptyset$ , respectively. The global transition rule will find a substitution unifying the two free variables  $y$  and  $z$ . Whether these two local transitions combine into an actual synchronised Meet action depends on the semantic variant employed for the global transition rule for action execution. Recall that we introduced three variants in Section 5.4.3. Suppose we use the third variant. Then, according to the first condition, the domain of the ground substitution  $\theta$  associated with the global transition rule must be the set of free variables in the actual parameters of the actuators, which in this case is the set  $\{y, z\}$ . This condition also states that  $\theta$  is a unifier of the actual parameters. As both agents use a free local variable, many unifiers are possible. A unifier of the action parameters is  $\{y := d, z := d\}$ , where  $d$  is any specific date. The second condition of the transition rule for group action execution states that the actual action parameter after substitution has to satisfy the aggregate constraints of both agents. To be specific, if the date substituted for  $y$  and  $z$  is  $d$ , then the second condition in this specific situation is:

$$(g = \{\text{Van der Broek, Meyer}\} \wedge v_1 = 29-1) \wedge (v_1 = d) \wedge (g = \{\text{Van der Broek, Meyer}\}) \not\vdash \perp.$$

The only value for  $d$  which satisfies this, is 29-1. So, there is only one substitution  $\theta$  possible, namely the substitution  $\{y := 29-1, z := 29-1\}$ . The third condition of the global transition rule (uniqueness of the unifier) is also satisfied. This means that the agents meet on 29-1.

The Meeting will also take place in the first semantic variant of group action execution; in the second variant, it won't.

The above programs are oversimplified, for expository reasons. For example, if the agents initially have inconsistent demands, then their resulting constraint stores will contain  $\perp$  for Meet. But as every formula is a logical consequence of  $\perp$ , the test  $v_1 = x$  (or  $v_1 = w$ ) would succeed for any value for  $x$  (or  $w$ , respectively). So, the agents should test for  $\perp$  first, in a realistic multi-agent program where the agents have no foreknowledge of the demands the other agent is going to make.

The reader might wonder how the negotiation phase is programmed in a realistic agent system, as in the example above both agents seem to 'know' which communication steps are expected. As agents are supposed to be autonomous, it might seem strange that agents anticipate each other's communication moves.

To be specific, when the agents Van der Broek and Meyer start their negotiation phase, they synchronously execute their `CommGroup` statements, with the action `Meet` as the action discussed. Only if both agents synchronously perform `CommGroups` about the same action, the negotiation can start. There isn't an initiator of the conversation; both agents act simultaneously. It all seems a bit too symmetrical to be realistic.

There are several responses to these questions. In the first place, GrAPL is an *abstract* programming language for group coordination. Thus, the statements for communication and action execution are on a high abstraction level. In an implementation of GrAPL, the synchronous primitives are likely to be implemented using a series of asynchronous communication primitives. So, on a lower abstraction level, the agents send messages to and fro according to a lower-level communication protocol which has an initiating agent and isn't symmetrical. The implementation underlying `CommGroupAdd` and `CommGroupReset` statements might be like this: the first agent arriving at a `CommGroup` statement on a certain action `a` broadcasts its initial demands to all agents that aren't excluded from the group of communicators by its own constraints. If there are agents in this group of potential negotiation partners which are willing to communicate about this action (which means that they also have a `CommGroup` statement with the action `a` as second parameter), then these agents all send their constraint on the parameters of `a` to the initiating agent. This agent waits for incoming messages with demands on `a` until enough agents have reacted to form a group of communicators for this communication round which satisfies the demands of all members of this group. Then, the initiating agent sends a message to the agents in this communicating group containing the resulting constraint (which is the conjunction of the constraints of the communicators).

The advantage of using the abstract synchronous statements of GrAPL over the asynchronous statements of the underlying communication protocols is that the program code for multi-lateral negotiation and group action execution can be more compact. The agent metaphor has been claimed to provide powerful abstract notions that can aid in the construction of the complex software needed nowadays, and GrAPL is an agent programming language in this spirit.

Returning to the abstraction level of GrAPL, we go into the use of the coordination primitives of GrAPL to construct program code for negotiation. In general, there are two ways to do this. The programmer can write *negotiation protocols* in GrAPL, or he can choose for *ad hoc* negotiation. In the second case, it is unclear how the conversation between negotiating agents will take place, so the agent program should contain many branches to take care of the different scenarios. When an agent reaches a `CommGroup` statement, it tries to

execute it, but if there is no suitable group of communicators which synchronises with the agent, then the agent waits until such a group does arrive (see the global transition rule for group communication; if there is no suitable group of communicators, no global transition is yielded and execution stalls for this branch of the agent program). During the negotiation phase, the agents can test the constraint on the action, to find out whether the negotiation has already converged to definite action parameters or whether the negotiation has led to conflicting demands on the action. Depending on the outcome of these tests, the agents decide whether they will try to communicate again and in which way (weaker or stronger demands), or to stop negotiating and start the execution phase. Example 5.14 shows an example of agent coordination without negotiation protocols. In case the agents use negotiation protocols, then the program code is less messy. As all agents are assumed to use the same protocol, each agent knows which synchronisation points to expect, and thus less branching is required. We will show an example of a coordination protocol for a group action next. In order to write down the protocol, we assume GrAPL includes recursive procedures.

**EXAMPLE 5.13** (A coordination protocol)

We give a negotiation protocol, for groups of benevolent agents discussing a future group action. We first show the protocol, and then explain it:

```

NegotiationProtocol(a,  $\psi(g)$ ,  $\varphi_{min}(\bar{v})$ ,  $\varphi_{sat}(\bar{v})$ ,  $\varphi_{opt}(\bar{v})$ ) :
CommGroupReset( $\psi(g) \wedge \varphi_{min}(\bar{v})$ , a);
if  $\perp$  for a
then ( $\psi'(g) := Adjust(\psi(g))$ ;
      NegotiationProtocol(a,  $\psi'(g)$ ,  $\varphi_{min}(\bar{v})$ ,  $\varphi_{sat}(\bar{v})$ ,  $\varphi_{opt}(\bar{v})$ ))
else (CommGroupAdd( $\varphi_{sat}(\bar{v})$ , a);
      if  $\perp$  for a
      then CommGroupReset( $\psi(g) \wedge \varphi_{min}(\bar{v})$ , a)
      else (CommGroupAdd( $\varphi_{opt}(\bar{v})$ , a);
            if  $\perp$  for a
            then CommGroupReset( $\psi(g) \wedge \varphi_{min}(\bar{v}) \wedge \varphi_{sat}(\bar{v})$ , a)
            else skip));

```

The negotiation procedure takes five parameters, namely the action discussed (a), a demand on the group composition ( $\psi(g)$ ), the minimal demand of the agent on the formal action parameters  $v_1, v_2, \dots, v_k$ , where  $k$  is the arity of the action ( $\varphi_{min}(\bar{v})$ ), a demand on the formal parameters that formalises what the agent would like best ( $\varphi_{opt}(\bar{v})$ ) and a demand on these parameters that formalises a satisfactory intermediate solution ( $\varphi_{sat}(\bar{v})$ ).

The agent first tries to find a group satisfying its group demand  $\psi(g)$ , which agrees with its minimal demand on the action parameters. The agent waits until it can communicate with a group satisfying  $\psi(g)$ . If this group doesn't agree with the minimal demand on the action parameters, then the first test for  $\perp$  succeeds, and the agent alters its constraint on the group composition (we don't go into details of the procedure *Adjust*). The recursive call to *NegotiationProtocol* results in an attempt to agree on the minimal parameter demands with a new group. As soon as there is a group that agrees with the minimal demands (and has minimal demands that this agent agrees with), then the agent tries to constrain the action parameters further according to its preferences. It starts carefully, by adding  $\varphi_{sat}(\bar{v})$ . If the others don't agree with this ( $\perp$  for  $\mathbf{a}$ ), then the agent resets the constraint to the previous parameter constraint, on which the agents agreed. As the other agents do the same thing, this group communication will always succeed, and the protocol is finished. If there is agreement in this round, the agent tries strengthening the constraint on  $\mathbf{a}$  by adding  $\varphi_{opt}(\bar{v})$ . The other agents also follow the protocol, so they do the same. If this succeeds, then the protocol is finished; otherwise, the agents reset the constraint to their previous agreement.

Note that we assume that the group of agents negotiating is fixed during execution of one call of *NegotiationProtocol*. This can be achieved by always using a  $\psi(g)$  of the form  $g = J$ , with  $J$  a set of agent names.

In the next example, we show agents which discuss an action without a protocol. To keep the example reasonably brief, the programs don't contain alternative branches for each possible flow of the conversation. The example illustrates that the agents strengthen their demands when there are no conflicts and there is no definite agreement yet, and that they weaken their demands when they have disagreements. We assume we use the first or third semantic variant for group action execution.

#### EXAMPLE 5.14 (Arranging a dinner date)

Two agents, Martha and Matthew, negotiate the time they will have dinner together. They definitely want to dine with each other, so each of these agents has the constraint that the other agent has to be part of the group performing the dine action. They don't agree yet on the precise time of their dinner. During the negotiation process, the demands of the agents can turn out to be inconsistent. To solve this, at least one of the agents has to weaken its demands. It can also happen that the aggregate constraints are still rather weak and don't fix one specific time for the dinner. Then, the

agents can strengthen their demands.

The dine action is unary; its sole explicit argument is the time. This is the program of Martha:

01. CommGroupReset( $v_1 \leq 19.00 \wedge \text{Matthew} \in g$ , dine);
02. if  $\perp$  for dine
03. then (CommGroupReset( $v_1 \leq 20.30 \wedge \text{Matthew} \in g$ , dine);
04.     if  $v_1 = t$  for dine
05.     then dine( $t$ )
06.     else dine( $u$ ))
07. else if  $v_1 = w$  for dine
08.     then dine( $w$ )
09.     else (CommGroupAdd( $v_1 = 19.00$ , dine);
10.         dine(19.00))

We number program lines for ease of reference. After the first CommGroup-statement, the program tests whether the resulting constraint set is inconsistent. Inconsistency results if the demand Matthew communicated is irreconcilable with Martha's demand. If so, then Martha weakens her constraints in a new communication attempt (line 03). As the inconsistent constraint has to be overwritten, a CommGroupReset is needed here; a CommGroupAdd would just add constraints to the already inconsistent store. The subprogram of lines 04–06 tests whether  $v_1 = t$  can be derived from the result of this communication. If this is the case, the agents agree on *one* precise time, which is bound to the variable  $t$ . Then, Martha goes to dinner at time  $t$ . If not, she leaves the choice up to Matthew (through the free variable  $u$ ). If the outcome of the first communication action is not inconsistent, the else-branch of line 07 is taken. The constraint resulting from the first CommGroupReset is tested. If this constraint is not strong enough to fix one definite time for the dinner, Martha communicates again. Now, a CommGroupAdd is appropriate, because the earlier constraint on dine has to be kept and strengthened.

Now, this is the program of Matthew:

01. CommGroupReset( $v_1 \geq 18.00 \wedge \text{Martha} \in g$ , dine);
02. if  $\perp$  for dine
03. then (CommGroupReset( $v_1 \geq 18.00 \wedge \text{Martha} \in g$ , dine);
04.     ? $(v_1 \leq x \wedge \exists y < x : v_1 \leq y$ , dine); dine( $x$ ))
05. else if  $v_1 = y$  for dine
06.     then dine( $y$ )
07.     else (CommGroupAdd( $v_1 \geq 19.00$ , dine);
08.         dine( $z$ ))

Matthew wants to dine after 18.00. If this proposal is not accepted, he tries to persuade Martha to give in by repeating his proposal of dinner after 18.00. In line 04, Matthew's constraint on dine is tested in a quite subtle way. We will come to this later. If the first proposal is accepted, Matthew tests whether one definite time is agreed upon. If not, he strengthens his constraints.

When these programs are executed, the first CommGroup action of the agents yields the consistent constraint  $18.00 \leq v_1 \leq 19.00 \wedge \text{Matthew} \in g \wedge \text{Martha} \in g$ . Martha proceeds with line 07, and Matthew with line 05, in which they both test their resulting constraints (stored locally now) for definiteness. Because the constraint still allows a range of times, the agents communicate again. The constraint resulting from this communication is  $v_1 = 19.00 \wedge \text{Matthew} \in g \wedge \text{Martha} \in g$ . The agents dine at 19.00.

But Martha and Matthew may not be the only agents around. Suppose there is another agent, Lucy, who would like to join Martha and Matthew. These agents haven't forbidden other agents to join the dinner; they just demanded that both of them should be present. Here is Lucy's program:

```

01. CommGroupReset( $g = \{\text{Matthew, Martha, Lucy}\} \wedge v_1 \geq 21.00, \text{dine}$ );
02. if  $\perp$  for dine
03. then(CommGroupReset( $g = \{\text{Matthew, Martha, Lucy}\} \wedge$ 
                                 $v_1 \geq 20.00, \text{dine}$ );
04.     dine( $s$ ))
05. else dine(21.00)

```

If the agent programs are executed now, then the first lines of the three programs can synchronise. This results in an inconsistent constraint ( $v_1 \leq 19.00 \wedge v_1 \geq 21.00 \vdash \perp$ ). So, all programs continue at line 03. Communication of the three agents results in the constraint  $g = \{\text{Matthew, Martha, Lucy}\} \wedge 20.00 \leq v_1 \leq 20.30$ . Lucy then tries to execute  $\text{dine}(s)$ , but she has to wait for the other two agents to arrive at the dine-statement. Martha first finds out that the constraint is not yet definite (line 04), and then proceeds to  $\text{dine}(u)$ . Note that Martha and Lucy both use free variables; this means they want Matthew to pick the definite time, as long as it is within the constraints they agreed upon. Matthew picks this time by testing the constraint in line 04. The formula tested states that  $x$  is the smallest upper bound on the time of dinner. The outcome of this test is that  $x$  is bound to 20.30. Matthew then executes  $\text{dine}(20.30)$  and the dinner party for three takes place.

## 5.6 Conclusions

We proposed a constraint programming language for agents with novel primitives for group communication and group cooperation. The statements for group communication are very expressive and allow groups of agents to negotiate over the conditions of execution of future group actions. These conditions are constraints on the composition of the group of participants and the parameters of the action. Communication is synchronous, allowing the dynamic formation of groups. In a language with only bilateral communication, programming a negotiation phase would be much more involved than in the language introduced here. If agreement is reached in the negotiation phase, the constraints agreed upon monitor the execution of the action discussed. The agents have to stick to their word; only then, group action can be successful. This way, social cohesion is enforced. Action execution is also synchronous, which is an intuitive manner to implement group action.

Our programming language only provides primitive means for negotiation. More sophisticated negotiation protocols or mechanisms can be programmed in GrAPL.

In an implementation of GrAPL, a constraint solver is built in. The underlying constraint solver accumulates the constraints of the communicating agents, computes the resulting constraint and delivers this to the communicating agents. The constraint solver makes communication of constraints possible, even if the number of communicating agents is large (imagine going to lunch with eighty agents), as in the implementation of GrAPL the agents send the constraints to the constraint solver, instead of to all other agents involved. One of the benefits of our approach is that we use results from another research area (constraint solving) to construct an agent programming language. In order to implement GrAPL, a suitable constraint solver can be selected and customised to the domain of group action parameters.

Many coordination problems in agent systems are about finding a solution on which all agents agree in some solution space; constraint solving is especially apt for this. A successful application of constraint-based approaches in artificial intelligence depends on suitably encoding the problems into constraints. But proving that this is possible for any coordination issue agents could encounter doesn't yield a practical coordination language. As we want to focus at applicability, and constraint programming and solving have proven their practical worth, we believe GrAPL is a significant contribution.

We provided the language with a well-defined operational semantics, built in a similar manner as the languages [12] and [59]. This gives the language a clear and unambiguous meaning, and offers a solid basis for building an im-

plementation. Such an implementation involves the development of suitable constraint solvers which is one of the main topics of future research.

The gap between theory and practice in agent research is a matter of concern for several researchers, including ourselves (see [18, 29, 55] for three contributions on these issues in CEEMAS 2001). By providing GrAPL with a formal semantics, we have created an agent programming language which is theoretically sound, and which can be implemented to build practical agent applications. GrAPL doesn't feature joint mental attitudes frequently used in logical theories on coordination ([33, 83], and more recently [28, 36]), as we didn't focus on the mental aspects of group collaboration. We do think that these aspects are important, and that integration of joint mental attitudes with the programming constructs of GrAPL is a promising direction for further work.

When we compare GrAPL with the skeleton programming language from the previous chapter, then GrAPL is more abstract. In the previous chapter, our aim was to give a realistic model of agent interaction, that doesn't abstract away from possible interaction problems. So, the semantic model of the previous chapter was rather detailed, including events taking place in the world, actions with duration and a model of real-time that allows different individual actions and group actions to be executed during overlapping time frames. Communication was asynchronous; agents send messages to each other, which they can pick up later by testing their sense buffers. In the previous chapter, we had both group actions and individual actions, but these were unparameterised and there were no specialised means to negotiate about the group performing the action. In this chapter, we abstract from real-time interaction issues, such as interacting actions and events disturbing action executions, and instead we focus on providing synchronous communication primitives for negotiating about implicit and explicit action parameters and on synchronous action execution. So, actions do have parameters in this chapter. The semantic model here is simpler; individual actions are interleaved, and group actions and communication are performed synchronously. We don't have any model of real-time. This higher abstraction level allows us to focus on details of group action coordination and to provide powerful statements. The use of synchronous communication in this chapter is a natural consequence of the more abstract approach. Synchronous communication can be implemented using asynchronous communication; it abstracts from the sending of messages to and fro to establish transfer of the message.

The new primitives of GrAPL, that is, the two forms of group communication and synchronised group action execution, are abstract and succinct means to construct programs of agents that interact with each other in order to first negotiate about group actions and later on execute them.

---

# An Abstract Coordination Language for Agents Interacting in Distributed Plan–Execute Cycles

---

*Don't worry about the future  
Or worry, but know that worrying is as effective as  
Trying to solve an algebra equation by chewing bubble gum*  
Mary Schmich, by way of Baz Luhrmann

## 6.1 Introduction

Agents are social, according to the most commonly accepted definition of agent-hood [130]. This means that agents communicate with each other, in order to exchange information and request and provide services. Agents can perform actions for or together with other agents, depending on their own motivations and available resources. These agent capabilities contribute to agents being *interacting* pieces of software. Agents pay attention to the actions of their fellow agents in the environment they inhabit, and sometimes try to persuade other agents to help them. In this chapter, we focus on agents that form

temporary alliances to jointly achieve common goals.

Collaborative agents can cooperate to achieve a mutually beneficial goal by developing a plan together for this goal. This is clearly a form of *distributed planning*. We *abstract* from the specific planning algorithms the agents use (see [27, 78, 97, 103, 126]), and instead focus on coordination between the agents. We provide a *coordination language* and an *agent architecture* that *facilitate* distributed planning and synchronised execution of the resulting plan. A coordination language provides statements for coordination of agents, *abstracting* from the internal make-up and processes of the agents. Aspects of agents like reasoning or message passing should be programmed in an ‘ordinary’ programming language, like AGENT-0 [113], 3APL [59] or JAVA. The statements of the coordination language should also be implemented in these lower level languages. The statements of our language enable agents to *form groups* dedicated to constructing a distributed plan together, to *communicate* about details of the plan (as produced by the planning algorithms used by the agents), to jointly decide that the plan is sufficiently elaborated by *committing* to it, and finally to *execute* it. The benefit of using a separate coordination language is that it focuses on one aspect of agents, namely coordination, and defines primitives for this in a clean and abstract manner, without distractions caused by details of internal agent statements. A coordination language can be used as a *wrapper*, to go around agents written in different programming languages. In the program of an agent, statements from the coordination language appear among statements from the programming language used for the agent. The primitives of the coordination language enable the agents to *interact* with each other in order to form and perform plans. This way, heterogeneous agents can coordinate their activities. The coordination language provides a neat separation of concerns, and enables reuse.

Our approach is based on *constraint programming* [111]. In constraint programming, indeterminacy about the value of a variable is gradually resolved by agents adding constraints on the value of this variable. In our approach, a *plan* is a named tuple consisting of several elements, like the set of actions, the order of these actions and the set of agents involved in execution of the plan. By establishing constraints on these components, the plan is formed. A *constraint solver* [119] is used to check whether the constraints of the agents forming the plan are consistent. After the negotiating agents have committed to the plan, the plan might still be partial, and the constraint solver uses its built-in planning knowledge to arrive at a sufficiently specified plan.

The starting points of this new coordination language are the same as those of the programming language GrAPL (described in Chapter 5): we use constraints, and the synchronous primitives of our coordination language are sim-

ilar (to a large extent) to the statements of GrAPL. But as we widen the scope of constraints from single actions to entire plans, there are many new aspects to be dealt with.

The set of actions of a plan can contain individual actions and group actions as well as see-to-it actions. These last actions simply specify that a certain goal is to be achieved, without details on how to go about this. When a plan is executed and a see-to-it action is encountered, a plan for the goal has to be formed and subsequently executed. So, a plan doesn't need to be fully elaborated before execution. Plans like this, which are gradually refined by introducing subgoals, are called *hierarchical plans*; hierarchical planning is also used in [103, 118]. Hierarchical plans have the advantage that part of the plan formation can be postponed and thus can take into account more up-to-date environment circumstances.

For each goal the agents want to achieve, there are two phases: the *plan creation phase* and the *plan execution phase*. In the plan creation phase, there is a group of *negotiator* agents, that communicate with each other, using a constraint store, to find a plan for the given goal. This means that the agents pick a plan name, and then start inserting constraints on the actions, the agents performing the actions and the order of the actions of that plan. For each activated plan name, there is a constraint store which is shared by the group of negotiator agents, but not accessible to agents outside this group. This is a departure from the principles of GrAPL, because there we only had local agent constraint stores. Here, we have a kind of global store, only accessible to a particular group (we call this semi-global). In Section 5.2 in the previous chapter, we offered several arguments in favour of using local constraint stores instead of global ones. Here, we choose to have global constraint stores, because a distributed plan by its nature is a global notion. A plan can be compared to a treaty between autonomous parties, which will govern the future behaviour of the parties. Like treaties, distributed plans are independent concepts that surpass the local agents that form and execute the plan. Therefore, it is intuitively obvious that plans in creation or execution are represented using a constraint store which is semi-global, instead of using local constraint stores. When the negotiators have inserted all of their constraints into the store of the plan, they commit to the plan, and the *execution phase* can start. Because the plan can contain synchronised group actions, it is useful to have a *plan controller* that decides on the details of plan execution. Each plan in execution has its own associated plan controller. This controller has authority over the actors of the plan; when it tells certain agents to synchronously perform an action, they have to do this. When the next action to be executed is a see-to-it action, the plan controller suspends and a new plan-execute cycle is started for the new goal.

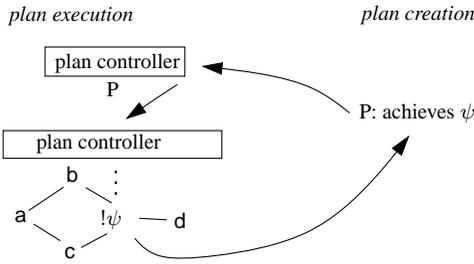


Figure 6.1: The plan–execute cycle

After a group of agents has formed a plan to obtain the goal and has executed it, the plan controller resumes with the rest of the outer plan. This way, the agents execute *distributed plan–execute cycles*.

It might seem that a global plan with a plan controller that dictates agent behaviours conflicts with agent autonomy, but as the plan being formed is agreed upon by the negotiating agents, these agents later on are obliged to honour the agreement. However, it is possible that some actors of the plan weren't negotiators, and thus didn't agree on their role in the plan. We will come back to this issue later on.

In Figure 6.1, we depict a plan–execute cycle. There is an outer plan (the partially ordered plan with actions  $a$ ,  $b$ ,  $c$ ,  $d$  and  $!\psi$ ) which is in execution; it has an associated plan controller, which already has taken care of the execution of  $a$ ,  $b$  and  $c$ , and is currently suspended at action  $!\psi$ . The goal  $\psi$  is broadcast to the agents, and some of them react by starting a new plan–execute cycle. They develop a plan  $P$  that will lead to  $\psi$ , and execute it. After the plan controller of this plan finishes, the controller of the outer plan takes over again. If plan execution finishes successfully, the plan controller terminates and a new planning phase can start. If actions fail due to synchronisation problems or environment conditions, the plan controller first informs the negotiators of the plan about the failure, such that they can decide to replan. After this, the plan controller terminates just as in the case of successful plan execution.

Thus, we propose a *generic architecture* for plan formation and execution. This architecture contains the agents, which form and execute plans, the constraint solver with the constraint stores of the plans, and the plan controllers. The statements of the coordination language enable the agents to interact with each other, update the constraint stores that represent plans being created, and start plan controllers. The constraint solver contains additional planning knowledge and can thus detect errors the agents make in their planning process and

complete the plan when the agents have inserted their constraints. The set of plan controllers is dynamic; whenever a new plan starts execution, a new plan controller springs up, which can suspend when a sub-plan has to be formed and which disappears when the execution of the plan is finished.

In the next section, we go into the nature of plans and the constraint stores that contain constraints on plans. In Section 6.3, we formally define the syntax and semantics of our coordination language. Section 6.4 provides an extensive example, illustrating the use of the coordination language. In Section 6.5, we describe the coordination architecture. Finally, Section 6.6 concludes this chapter.

## 6.2 Plan features and constraint stores

Underlying the definition of both plans and statements of the coordination language are these sets:

- \*  $\mathcal{A}$  = the set of atomic physical actions. For simplicity, we focus on unparameterised actions here.
- \*  $\mathcal{I}$  = the set of agent identities.
- \*  $\mathcal{PN}$  = the set of plan names.
- \*  $\mathcal{PV}$  = the set of plan variables; each agent has at its disposal its own plan variables, a subset of  $\mathcal{PV}$ .
- \*  $\mathcal{LV}$  = the set of local agent variables.
- \*  $\mathcal{L}$  = a predicate logical language used to formalise constraints on plans. We assume  $\mathcal{L}$  is equipped with a consequence relation denoted by  $\models_{cs}$  (cs abbreviates Constraint Solver). As in previous chapters,  $free(\varphi)$  is the set of free variables in the formula  $\varphi$ .

As stated in Definition 6.1 below, actions from the set  $\mathcal{A}$  form the basis of plans. To keep things manageable, we look at unparameterised actions in this chapter, in contrast to the actions in the previous chapter. In the concluding section, we will briefly discuss a combination of the coordination language with GrAPL, in which actions are parameterised.

The allocation of actions to particular agents or groups of agents is part of the planning process. The task allocation determines whether a certain action will be an individual action or a group action. Of course, some actions from

$\mathcal{A}$  can only be allocated to one agent, while other actions always must be performed by a group. For example, in the domain of cleaning office buildings, the action of vacuuming a room is an individual action, the action of moving a heavy cupboard is a group action, and the action of cleaning the windows can be both individual or collective, depending on the amount of glass present in the office.

The sets  $\mathcal{PV}$  and  $\mathcal{PN}$  are closely related. Both plan variables and plan names are used to refer to different plans. At each moment during execution of a multi-agent system, there can be various plans under construction and in execution, by disjoint or even overlapping groups of agents (see also Section 6.4). Plan variables from  $\mathcal{PV}$  are assigned values from the set of plan names  $\mathcal{PN}$ . When including a coordination statement in an agent program, the programmer should use a plan variable to refer to a plan and its associated constraint store. When the coordination statement is executed, and there is not yet a binding to the plan variable, the system chooses a new, unused plan name and binds this to this variable. The programmer is not allowed to choose a plan name himself, because he could choose a plan name already used by another agent for another plan, and thus create confusion. Example 6.3 later on will clarify this. The set of plan variables is divided into disjoint subsets, such that each agent has its own private plan variables. The use of these sets will also become apparent in Example 6.3. The set  $\mathcal{LV}$  contains local variables that agents can use to test the constraint store of a plan (named *plan store* from now on).

Our notion of plans is very close to the notion of *operators*, used in e.g., [118]. Though the definition of operators differs, an operator usually contains a precondition, a postcondition and a plan body. The precondition specifies environment circumstances in which the plan body can be executed, with the goal of achieving the postcondition. The plan body specifies the actions and the order in which they need to be done. In our architecture, the plan body is represented by a partially ordered set of actions. These actions can be atomic actions from  $\mathcal{A}$  or actions of the form  $!\psi$ . This should be read as “see to it that  $\psi$  becomes true”. Actions like these allow hierarchical planning. If a plan contains  $!\psi$ , then a sub-plan to achieve  $\psi$  is made when  $!\psi$  is executed, and not earlier. This is useful, because it enables agents to adjust the plan to the circumstances in the environment close to the time of execution.

**DEFINITION 6.1** (Plan components)

The set  $\mathcal{PC}$  of plan components is defined:

$$\mathcal{PC} = \mathcal{A} \cup \{!\psi \mid \psi \text{ is a closed formula of } \mathcal{L}\}.$$

So, a plan can have two different kinds of actions: basic actions, which are simply executed, and see-to-it-actions, where a whole new planning and exe-

cution phase are started in a recursive manner. As the actions in the set  $\mathcal{A}$  are physical in nature, we don't have mental actions in the plans. In the terminology of Chapter 4, plan execution is an *interaction* with the environment. The coordination language entirely abstracts from internal agent reasoning, called *intraaction* in Chapter 4. The coordination language enables the agents to communicate with each other and execute plans, which both are classified as *interaction*.

As described in the introduction, a plan is formed by inserting constraints into the constraint store associated with the plan name. The constraints pertain to certain features of the plan, such as the action set or the task allocation. Each plan feature is represented by a special variable.

DEFINITION 6.2 (Plan features)

We define six *plan features*

$[Purpose, Circumstances, Actions, <, Negotiators, Actors]$ ,

where *Purpose* and *Circumstances* are variables of type  $\mathcal{L}$ , *Actions* is a variable of type  $\wp(\mathcal{PC})$ ,  $<$  is a variable of type  $\wp(\mathcal{PC} \times \mathcal{PC})$ , *Negotiators* is a variable of type  $\wp(\mathcal{T})$  and *Actors* is a variable of type  $\mathcal{A} \dashrightarrow \wp(\mathcal{T})$  (the set of partial functions from  $\mathcal{A}$  to  $\wp(\mathcal{T})$ ).

The six features of a plan all are *variables*, which gradually are constrained during the plan creation phase. To be precise, the values of the plan features *Purpose*, *Circumstances* and *Negotiators* are determined at the start of the planning phase, when the group of negotiators is formed. The other three variables are worked out during the planning phase. *The name of the plan actually doesn't refer to a plan, but to the constraint store containing the demands on the plan.* We will often be sloppy with this, and talk about “the plan called *P*”. During the plan creation phase, the constraints in the plan store allow a set of possible plans. Example 6.1 below illustrates this. At the end of the planning phase, when the agents have committed to the plan, the plan is ready and *definite*. All six plan components then have been completely determined, that is, the constraints imply that *Actions* is a specific set, *Purpose* is equivalent to a ground formula from  $\mathcal{L}$ , etc. As the six variables each have a definite value, the space of possible plans has been narrowed down to one.

A *plan* thus is an association of values with a six-tuple of plan features. After the planning phase has ended, the formula associated with *Purpose* formalises the goal of the plan, and the value of *Circumstances* is a formula describing suitable environment conditions for the plan to be performed. These two formulas correspond to the postcondition and precondition of operators. The planning agents use these two formulas as an input to their planning rou-

tines, from which we abstract, and which yield contributions to the distributed plan under construction. *Purpose* and *Circumstances* can also play a role at the start and end of plan execution, to check whether proper *Circumstances* indeed are present in the world before execution, and to check whether the *Purpose* of the plan indeed has been achieved after execution. These checks are performed at the lower agent level, and so we don't go into them. The value of *Actions* in a finished plan is the set of plan components constituting the plan. Note that a committed plan is still partial, in the sense that it can contain see-to-it-statements! The value of  $<$  is a partial order on the action set. To ensure this, the consequence relation  $\models_{cs}$  used by the constraint solver has the properties of partial orders as axioms. The partial order fixes the execution order of some actions. Actions that are not ordered by it are independent and can be executed in parallel (according to the planning agents). The value of *Negotiators* is the set of agents which have agreed to form the plan together, and the value of *Actors* is a function, associating a group of agents to each atomic action in the plan. Actions which have been allocated to a group containing more than one agent must be done by these agents in a synchronised manner. So, the value of *Actors* describes the *task allocation* of the plan.

Plan features of a certain plan fulfil the same role as the formal parameters  $v_1, \dots, v_k$  and  $g$  of a certain action in GrAPL. A plan can be seen as a composite action, with formal parameters that determine the exact nature of the action. These formal parameters are the plan features. Just as in GrAPL, we always use the same variables for the parameters of any plan, namely the six plan features introduced above. Like in GrAPL, the agents communicate to establish constraints on the parameters of the plan. Differences with GrAPL are that we have semi-global (accessible to the *Negotiators* only) constraint stores here, and that a plan can only be executed when the formal parameters are definite. In GrAPL agents have local constraint stores and actions can be executed when the constraints of the actors don't fix a specific value for each action parameter.

During the plan creation phase, the plan is partial, because the six variables are only determined by the constraints given by the planning agents (the *Negotiators*), and generally these won't completely fix the values of the variables during the planning process. The language  $\mathcal{L}$  for phrasing constraints at least contains set-theoretic operations (e.g.,  $\in$ ,  $\subseteq$ ) among its predicates. Additionally, there can be predicates expressing characteristics of agents. For example, the constraint  $reliable(\text{James}) \leftrightarrow \text{James} \in \text{Actors}(\text{transport\_diamond})$  demands that James should be involved in transporting a certain diamond if and only if he is a reliable agent. Typically, during the plan creation phase we will have constraints like  $A \subseteq \text{Actions}$ , where  $A$  is a set of plan components. So, other plan components could also become part of the plan later on. This is an

example of a plan store during the planning phase, specifying a set of possible plans:

EXAMPLE 6.1 (A plan under construction)

$$\begin{aligned}
 \textit{Purpose} &= \textit{room\_clean} \wedge \\
 \textit{Circumstances} &= \textit{room\_messy\_and\_dirty} \wedge \\
 \textit{Negotiators} &= \{\textit{James}, \textit{Martha}\} \wedge \\
 \{\textit{!floor\_very\_clean}, \textit{empty\_ashtray}, \textit{open\_windows}\} &\subseteq \textit{Actions} \wedge \\
 \textit{open\_windows} &< \textit{!floor\_very\_clean} \wedge \\
 \textit{Actors}(\textit{empty\_ashtray}) &= \{\textit{John}\}
 \end{aligned}$$

Note that in the example above, the set of actions is not fixed yet and the allocation and the ordering of the actions is only partial.

After the negotiators have committed to the plan and the plan is finished by the constraint solver, the constraints allow only one plan. This is an example of a finished plan:

EXAMPLE 6.2 (A definite plan)

$$\begin{aligned}
 \textit{Purpose} &= \textit{room\_clean} \wedge \\
 \textit{Circumstances} &= \textit{room\_messy\_and\_dirty} \wedge \\
 \textit{Negotiators} &= \{\textit{James}, \textit{Martha}\} \wedge \\
 \textit{Actions} &= \{\textit{!floor\_very\_clean}, \textit{empty\_ashtray}, \\
 &\quad \textit{open\_windows}, \textit{!}\neg\textit{room\_messy}\} \wedge \\
 \textit{open\_windows} &< \textit{!}\neg\textit{room\_messy} \wedge \\
 \textit{!}\neg\textit{room\_messy} &< \textit{!floor\_very\_clean} \wedge \\
 \textit{Actors}(\textit{empty\_ashtray}) &= \{\textit{John}\} \wedge \\
 \textit{Actors}(\textit{open\_windows}) &= \{\textit{Mary}, \textit{Martha}\}
 \end{aligned}$$

Though this plan is definite, the order of the actions is still partial and not total.

To store the constraints on plans, the state of the system has to contain a plan constraint store association:

DEFINITION 6.3 (Plan constraint store association)

A *plan constraint store association* is a partial function  $\mu : \mathcal{PN} \dashrightarrow \mathcal{L}$ .

This function maps plan names to the present constraint on that plan. The function is partial, as only part of the plan names are in use for plans being formed or in execution.

We also provide a definition that formalises that a plan store is *definite*, which is the case when the constraints on the plan allow only one value for each of the six plan features.

DEFINITION 6.4 (Definiteness of a plan store)

A plan store  $P \in \mathcal{PN}$  is *definite* in the context of a constraint store association  $\mu$  when the following holds:

- \* There is a closed formula  $\varphi \in \mathcal{L}$ , such that  $\mu(P) \models_{\text{cs}} \text{Purpose} = \varphi$ .
- \* There is a closed formula  $\psi \in \mathcal{L}$ , such that
$$\mu(P) \models_{\text{cs}} \text{Circumstances} = \psi.$$
- \* There is a set of plan components  $A \subseteq \mathcal{PC}$ , such that
$$\mu(P) \models_{\text{cs}} \text{Actions} = A.$$
- \* There is a relation  $R \subseteq \mathcal{PC} \times \mathcal{PC}$ , such that  $\mu(P) \models_{\text{cs}} < = R$ .
- \* There is a set of agents  $N \subseteq \mathcal{I}$ , such that  $\mu(P) \models_{\text{cs}} \text{Negotiators} = N$ .
- \* There is a partial function  $f : \mathcal{A} \dashrightarrow \wp(\mathcal{I})$  with domain  $A \cap \mathcal{A}$  (all atomic actions in the plan), such that  $\mu(P) \models_{\text{cs}} \text{Actors} = f$ .

This definition is related to Definition 5.9 in Section 5.4.3 in the previous chapter. When a plan store is definite, we often say that “the plan is definite”, and identify the constraints on the plan with an association of specific values with the plan features.

## 6.3 Syntax and semantics of the plan coordination language

After establishing this background, we define the statements of the coordination language:

DEFINITION 6.5 (Coordination statements)

The coordination language contains the following statements:

- \*  $\text{FormGroup}(\alpha, \omega, p, \varphi)$ , where  $\alpha, \omega, \varphi \in \mathcal{L}$  are closed formulas and  $p \in \mathcal{PV}$ .
- \*  $\text{CommGroup}(\varphi, p)$ , where  $p \in \mathcal{PV}$  and  $\varphi \in \mathcal{L}$  is a closed formula.
- \*  $?( \varphi, p)$ , where  $\varphi \in \mathcal{L}$  and  $p \in \mathcal{PV}$ .
- \*  $\text{Commit}(p)$ , where  $p \in \mathcal{PV}$ .
- \*  $\text{Execute}(p)$ , where  $p \in \mathcal{PV}$ .

To define agent programs, the statements above are combined with a lower level (agent) programming language. This programming language provides

statements for other agent capabilities, such as reasoning and ordinary communication. Also, this programming language provides control constructs, such as conditional branching, recursion, sequential composition and non-deterministic choice. Below, we explain the coordination statements and give their formal operational semantics.

The semantics we use is two-layered, like in the previous two chapters. As we only look at agent coordination and abstract from the local agent reasoning, we only need to give transition rules for the global level semantics of the coordination statements. We assume the underlying programming languages (or language) used to program the agents have well-defined local semantics.

In order to define global transition rules, we first define global system configurations. The global system configuration at each moment contains at least the local agent configurations, the set of plan controllers that are active at that time, the set of plans that are committed and ready but not in execution yet and a plan constraint store association:

**DEFINITION 6.6** (Global system configuration)

A *global system configuration* is a tuple  $\langle \dots, \{ \langle S_\iota, \pi_\iota \rangle \mid \iota \in \mathcal{I} \}, \chi, \rho, \mu \rangle$ .

Here,

- \*  $\langle S_\iota, \pi_\iota \rangle$  is a local agent configuration of agent  $\iota \in \mathcal{I}$ , where  $\pi_\iota$  is the local program left to be executed and  $S_\iota$  is the local agent state.
- \*  $\chi \subseteq \mathcal{PN}$  is the set of plans that are currently being executed.
- \*  $\rho \subseteq \mathcal{PN}$  is the set of plans that are ready for execution, but aren't in execution yet.
- \*  $\mu : \mathcal{PN} \dashrightarrow \mathcal{L}$  is a plan constraint store association.
- \*  $\dots$  are other global configuration elements which we abstract of, because these aren't relevant for plan coordination (for example, the state of the world).

We demand that  $\chi \cap \rho = \emptyset$ , and that for each  $P \in \rho \cup \chi$  it holds that  $P$  is definite.

The demand that  $\chi$  and  $\rho$  have no elements in common is made because a plan can't be in execution and waiting to be executed at the same time. The plans in  $\chi$  each have an associated plan controller, to which we also refer with the name of the plan. The plans in the sets  $\rho$  and  $\chi$  must be definite, because otherwise they are still too partial to be executed. The plan constraint store association  $\mu$  binds a constraint to each plan name which is in use by the system of agents. We largely abstract from the contents of the local agent configurations  $\langle S_\iota, \pi_\iota \rangle$ ,

as the details of the internal make-up of the agents belong to the lower agent level. We do explicitly represent the local agent program, as execution of coordination statements changes the program left to be executed.  $S_\iota$  represents the local agent state, which can for example consist of a belief base, a set of current intentions and a set of joint intentions. We sometimes use  $A_\iota$  as a shorthand for  $\langle S_\iota, \pi_\iota \rangle$ . The global configuration can contain additional elements from which we abstract, such as the state of the world. In the above definition, these elements are suggested by including  $\dots$  as an element of the global configuration. For readability and brevity, we leave out these dots in the sequel of this chapter.

In general, a global transition rule of a coordination statement  $C$  will look like this:

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{C(\text{args}_\iota)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi', \rho', \mu' \rangle, \text{ other info}}$$

Here,  $A_\iota$  abbreviates  $\langle S_\iota, \pi_\iota \rangle$ , and  $A'_\iota$  abbreviates  $\langle S'_\iota, \pi'_\iota \rangle$ . In this rule, a group of agents  $J$  synchronously executes the coordination statement  $C$ . Each agent has its own coordination parameters: agent  $\iota$  tries to execute  $C(\text{args}_\iota)$ . This coordination statement is part of the program of the agent, which is largely written in some other programming language. In order to be able to phrase global transition rules for the coordination statements, we have to make assumptions about their local semantics, even if we aren't looking at the local semantics of the programming language the rest of the agent program is written in. As can be seen above, we assume that local semantics of a coordination statement yields a local transition labelled with the coordination statement and its arguments,  $C(\text{args}_\iota)$ . By putting the coordination statements in the labels of local transitions, they are available for the global semantics to process. The meaning of the coordination statements is largely determined at the global level, though there can also be local effects. For example, when agents form a group to make a plan, they can establish a joint intention to achieve the *Purpose* of the plan to be formed. Whether this happens depends on the mental make-up of the agents. If a mental update is a result of executing `FormGroup`, this is encoded in the local semantics of this coordination statement, by transforming the local state  $S_\iota$  into  $S'_\iota$ . The local transitions of coordination statements always model successful executions of the coordination statements. Of course, the coordination statement can fail; the global semantics is defined in such a way that the local updates to the agent states aren't performed in case of failure.

On the basis of the information in the labels of the local transitions above the line of the transition rule, a global transition is constructed. This global transition transforms the old global configuration into the new one, and it can

yield additional information, such as a boolean value to indicate whether the coordination statement has succeeded. This information, which appears as “other info” in the template transition rule above, can be used by the local agent programs in the sequel of the execution. We don’t know the structure of the local agent states; this is the reason that we add the “other info” as the second result of a global transition (the first being the new global configuration). Depending on the design of the agents, this information will become part of the local agent states in some way or another.

There are two manners in which coordination statements can fail. In the first place, synchronisation can fail. This happens for the coordination statements that require the whole group of *Negotiators* to be present. When not all negotiators are synchronously performing their coordination statement, then no global transition is generated. This means that the global computation *blocks*, and the local agents have to *wait* until all *Negotiators* are present and the coordination statement can be performed. The global computation also blocks when an agent unsuccessfully tests a plan store. This doesn’t mean that the computation of the system is over; agents can take other branches of their local programs that don’t lead to deadlock. The second way of failing happens when agents try to form the plan by executing *CommGroups*. When the *Negotiators* propose constraints on the plan which aren’t consistent with each other or earlier constraints on the plan, then this type of failure occurs. The constraints are not added to the plan store, as this would result in the constraint  $\perp$  (inconsistency), and all earlier planning information would be lost. Instead, a global transition is generated that keeps the plan store unchanged, and returns a boolean value *false* to the negotiating agents. The agents thus can find out that the group communication has failed because of disagreement, and subsequently try other proposals in group communication.

Now, we come to the discussion of the statements of our coordination language. The statement  $\text{FormGroup}(\alpha, \omega, p, \varphi)$  informally translates into: “Form a group which will negotiate about a plan referred to by  $p$  aimed at realising  $\omega$  starting from situations where  $\alpha$  holds, and the group composition has to obey the constraint  $\varphi$ .” A *FormGroup* is synchronously performed by a group of agents. This is the semi-formal semantics of *FormGroup*:

Whenever a number of agents  $J \subseteq \mathcal{I}$  try to form a group by each ( $\iota \in J$ ) executing  $\text{FormGroup}(\alpha, \omega, p_\iota, \varphi_\iota)$ , then this will succeed if  $\bigwedge_{\iota \in J} \varphi_\iota \wedge \text{Negotiators} = J \not\vdash_{\text{CS}} \perp$ , that is, if the composition of the group is consistent with the conjunction of the demands on group composition. Note that the agents have to agree upon the purpose  $\omega$  and circumstances  $\alpha$  of the plan. If successful, the

system chooses a new, unused plan name  $P$  ( $P \notin \text{domain}(\mu)$ ), and passes this name to the agents, which associate the name with the plan variables they use ( $p_i, i \in J$ ). Also, the FormGroup-operation initialises the constraint bound to  $P$ , as the purpose, circumstances and negotiator group are fixed. Formally:  $\mu(P) := \text{Purpose} = \omega \wedge \text{Circumstances} = \alpha \wedge \text{Negotiators} = J$ . If the group of agents  $J$  doesn't satisfy the constraints of the agents, then execution blocks; no global transition is generated.

When a group is formed, only these agents are allowed to add constraints to the plan. A successful FormGroup-statement initiates a plan store for the new plan with *Purpose*  $\omega$  and *Circumstances*  $\alpha$ , and the fixed group of *Negotiators*  $J$ . All members of the group  $J$  have to be present when the plan is updated. This way, we can be sure that intruders from outside can't influence the plan and that the negotiators agree on the shared plan being formed. If group formation fails, because the present group doesn't satisfy the demands of the group members, then it is possible that there is another group (at the same time or later on) that is suitable. If there is no such group available, all agents wait at their FormGroup statements for a suitable group to arise, of which they may or may not be part.

Another result of FormGroup is that the new plan is *named*. This is important, because this name will be bound to the constraint store shared by the negotiators, that will gradually define the plan, and the name of the plan is needed to be able to execute it, and to phrase constraints on the plan. Though the name formally only refers to a constraint on the plan (through the plan constraint store association), we will also use the name to refer to the definite plan resulting at the end of the plan creation phase and to refer to the plan controller of the plan in the execution phase. The agents use different plan variables to refer to the plan, and a result of a FormGroup is that these variables can be bound to the chosen plan name by the local agents.

The FormGroup-statement doesn't correspond to any statement in GrAPL. This is because in GrAPL, the group of agents talking about constraints on a group action isn't fixed. This makes GrAPL quite flexible. Agents can come in at any time (as long as the constraint-under-construction on the action allows this), add some constraints, and then not take part in the negotiation anymore. Here, this would be peculiar, because we have a semi-global constraint store. This is necessary and intuitive, because the agents are constructing a *plan*, and it would be awkward to have this plan scattered in the local constraint stores of the agents. But then, if we wouldn't work with a fixed set of planners, the constraint store could be written by any agent that likes to, and there is no way to control the constraints written into the store. As the plan is shared, all

planning agents involved should agree on each constraint added. So, we need to know what the group of planners is, and for this, we need the FormGroup-statement. We are not alone in making the distinction between group formation and planning. For example, in [131], Wooldridge and Jennings also separate the stage of team formation from the stage of plan formation.

Now, we give the formal transition rules for FormGroup. Let  $J \subseteq \mathcal{I}$  be a non-empty set of agent names, let  $P \in \mathcal{PN}$  be a plan name, let  $A_\iota = \langle S_\iota, \pi_\iota \rangle$  and let  $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$ . This is the transition rule for FormGroup:

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{FormGroup}(\alpha, \omega, p_\iota, \varphi_\iota)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi, \rho, \mu' \rangle, P}$$

where the following conditions hold:

- \*  $P \notin \text{domain}(\mu)$
- \*  $\bigwedge_{\iota \in J} \varphi_\iota \wedge \text{Negotiators} = J \not\vdash_{\text{CS}} \perp$
- \*  $\mu' = \mu[\text{Purpose} = \omega \wedge \text{Circumstances} = \alpha \wedge \text{Negotiators} = J/P]$

This rule is just a formalisation of the informal semantics we gave earlier. The result of the global transition is a new global configuration, as well as the name  $P$  of the new plan in creation which is now available for the local agents to bind to their plan variables  $p_\iota$ . The local updates to the states of the agents as a result of forming a group (for example, a joint intention towards the *Purpose* of the plan) indeed take place; the local changes to the agent configurations ( $A_\iota \rightarrow A'_\iota$ ) are part of the change to the system configuration. The configurations of agents that don't participate in the group formation stay the same. Forming a group fails if the group composition is inconsistent with the demands of the group members, or these demands are inconsistent among themselves. No global transition results for this group  $J$  trying to initiate a plan creation phase. The local agent programs of the agents in  $J$  are stuck at the FormGroup statement. This situation remains like this until a suitable group of agents synchronously attempts to form a group. It seems likely that some agents from  $J$  won't be part of this new group.

To actually plan, we have CommGroup( $\varphi, p$ ), which informally translates to "Add the constraint  $\varphi$  to the present constraint on the plan referred to by  $p$ ". Here,  $\varphi$  is a formula stating demands of the agent executing the CommGroup on the properties of the plan (that is, on the actions in the plan, their order and the agents performing them). As CommGroup-statements must always be preceded by a FormGroup-statement, the plan variable  $p$  is already bound to a plan name

$P \in \mathcal{PN}$ . `CommGroup` is a synchronous communication statement, that can only be successful if all of the *Negotiators* of plan  $P$  execute it simultaneously. This is the semi-formal semantics of `CommGroup`:

Whenever a number of agents  $J \subseteq \mathcal{I}$  try to modify a plan by each executing `CommGroup`( $\varphi_\iota, p_\iota$ ), then this will succeed if all  $p_\iota, \iota \in J$  are bound to the same plan name  $P$ ,  $\mu(P) \models_{\text{cs}} \text{Negotiators} = J$  (that is, the proper group of agents present) and  $\mu(P) \wedge \bigwedge_{\iota \in J} \varphi_\iota \not\models_{\text{cs}} \perp$  (that is, the constraints of the agents are consistent with each other and the stored plan constraint). If successful, the operation updates the constraint bound to  $P$ . Formally:  $\mu(P) := \mu(P) \wedge \bigwedge_{\iota \in J} \varphi_\iota$ . Finally, the execution yields a boolean value, indicating whether the attempt to update the plan succeeded.

If the constraints of these agents are consistent, then the conjunction is added to the plan store. If not, then the plan store remains unchanged. The boolean is available for the agents to include in their local state; it can be tested by the agents involved to determine their next step.

The coordination statement `CommGroup` is similar to the statement `CommGroupAdd` from the programming language GrAPL from the previous chapter. `CommGroupAdd` also strengthens the current constraint by adding a new conjunct to the local constraints on a certain group action. But unlike the coordination statement `CommGroup`, `CommGroupAdd` doesn't fail when the proposals of the group members lead to inconsistency. Here, we choose to let `CommGroup` have no effect on the plan store in case of inconsistency of proposals, because otherwise one would lose all information about the plan in creation stored in the plan store, as it is overwritten with  $\perp$ .

We give the two global transition rules for `CommGroup`, for successful and failing group communication, respectively. With failing group communication, we refer to the situation that the proper group of *Negotiators* takes part in the group communication, but with proposals which aren't consistent with each other and/or the previous constraint in the plan store. For the situation that the proper *Negotiators* aren't present, no global transition results. Then, the agents which are present have to wait for the rest of the negotiating agents. We again assume that  $J \subseteq \mathcal{I}$  is a set of agent names,  $P \in \mathcal{PN}$  is a plan name,  $A_\iota = \langle S_\iota, \pi_\iota \rangle$  and  $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$ . We start with the rule for a successful `CommGroup`:

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{CommGroup}(\varphi_\iota, P)} A'_\iota}{\langle \{A_\iota \mid \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota \mid \iota \in J\} \cup \{A_\iota \mid \iota \in \mathcal{I} \setminus J\}, \chi, \rho, \mu' \rangle, \text{true}}$$

where the following conditions hold:

$$* \mu(P) \models_{\text{cs}} \text{Negotiators} = J$$

$$* \mu(P) \wedge \bigwedge_{\iota \in J} \varphi_{\iota} \not\models_{\text{cs}} \perp$$

$$* \mu' = \mu[\mu(P) \wedge \bigwedge_{\iota \in J} \varphi_{\iota} / P]$$

Again, we simply formalise the semi-formal semantics given earlier. The result of the global transition is a new global configuration, in which the local configurations of the communicators are modified and the plan store of  $P$  is updated, and the boolean *true* to signal that the group communication has succeeded.

This is the transition rule for an unsuccessful **CommGroup**:

$$\frac{\text{for all } \iota \in J : A_{\iota} \xrightarrow{\text{CommGroup}(\varphi_{\iota}, P)} A'_{\iota}}{\langle \{A_{\iota} \mid \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{ \langle S_{\iota}, \pi'_{\iota} \rangle \mid \iota \in J \} \cup \{A_{\iota} \mid \iota \in \mathcal{I} \setminus J\}, \chi, \rho, \mu \rangle, \text{false}}$$

where the following conditions holds:

$$* \mu(P) \models_{\text{cs}} \text{Negotiators} = J$$

$$* \mu(P) \wedge \bigwedge_{\iota \in J} \varphi_{\iota} \models_{\text{cs}} \perp$$

Because of the first condition above, the group of communicators must still be the group of *Negotiators* of the plan, as was the case for successful group communication. In case the second condition also holds, the demands of the negotiating agents are inconsistent with each other or with the previous demands on the plan. As a result, the plan store isn't changed and the boolean *false* is returned. The **CommGroup** statement is removed from the programs left to be executed of the participants ( $\pi_{\iota} \rightarrow \pi'_{\iota}$ ), but the update to the local agent states ( $S_{\iota} \rightarrow S'_{\iota}$ ) which was locally computed isn't performed. For example, if in a particular implementation of **CommGroup** an agent forms an intention towards each action it adds to the plan, this results in a transformation of the local agent state ( $S_{\iota}$  becomes  $S'_{\iota}$ ) when a proposed constraint of this agent implies that actions are added to the plan. If the **CommGroup** doesn't succeed, this intention shouldn't be established, so the local agent state stays  $S_{\iota}$ .

The **CommGroup**-statement is close to the **tell**-statement from constraint programming [111], in that it adds information to the constraint store. We also need an equivalent of **ask**, to test the constraint store. For this, we have  $?( \varphi, p )$ , with informal reading: "Test whether  $\varphi$  follows from the constraint on the plan referred to by  $p$ ". This is the semi-formal meaning of this statement:

Whenever an agent  $\iota \in \mathcal{I}$  tries to test the store of a plan by executing  $?( \varphi, p )$ , and  $p$  is associated with the plan name  $P$ , then

this will succeed if  $\mu(P) \models_{\text{cs}} \iota \in \text{Negotiators}$  (that is,  $\iota$  is one of the *Negotiators* of the plan) and there is a ground substitution  $\theta$ , with  $\text{dom}(\theta)$  equal to the set of free local variables in  $\varphi$ , such that  $\mu(P) \models_{\text{cs}} \varphi\theta$  (that is, the store of  $P$  entails this instantiation of  $\varphi$ ). If successful, the operation yields the substitution  $\theta$ , which represents information about the values of features of  $P$ . If not, the execution blocks.

For the definition of substitutions and their domains, we refer to the previous chapter, Subsection 5.4.1, Definition 5.6. Typical tests of a plan are  $?(\mathbf{a} \in \text{Actions}, p)$ , which tests whether  $\mathbf{a}$  is an action of the plan  $p$ ,  $?(\mathbf{a} < \mathbf{b}, p)$ , which tests whether action  $\mathbf{a}$  precedes action  $\mathbf{b}$ , and  $?(\text{Actions} = x, p)$ , where  $x$  is a free local variable and which tests whether the action set of  $p$  is fixed to one specific set. We will explain the semantics of this last statement in more detail. Suppose  $P$  is the plan name bound to  $p$ . The test succeeds if the testing agent is allowed access to the information about the plan  $P$  and if there is a substitution which binds a ground value to  $x$  such that the constraints on  $P$  entail that the set of actions of  $P$  equals this value. This implies that the ground value bound to  $x$  must be a set of plan components. If the test succeeds, that is, if indeed the action set of  $P$  has been constrained to one specific set of plan components, then this set is bound to  $x$ , such that the agent which performed this test can use this information.

We only have a global transition rule for successful tests. In case the test fails, no global transition results, which means that the local execution of the test blocks. Let  $\iota \in \mathcal{I}$  be an agent name, let  $P \in \mathcal{PN}$  be a plan name, let  $\theta$  be a ground substitution with  $\text{dom}(\theta) = \text{free}(\varphi) \cap \mathcal{LV}$ , let  $A_\iota = \langle S_\iota, \pi_\iota \rangle$  and let  $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$ :

$$\frac{A_\iota \xrightarrow{?(\varphi, P)} A'_\iota}{\langle \{A_\iota \mid \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota\} \cup \{A_\iota \mid \iota \in \mathcal{I} \setminus \{\iota\}\}, \chi, \rho, \mu \rangle, \theta}$$

where the following conditions hold:

- \*  $\mu(P) \models_{\text{cs}} \iota \in \text{Negotiators}$
- \*  $\mu(P) \models_{\text{cs}} \varphi\theta$

These conditions state that only *Negotiators* of a plan have access to its plan store, and that an instantiation of the formula tested ( $\varphi\theta$ ) follows from the plan store.

Next is  $\text{Commit}(p)$ , which informally reads “Complete the plan referred to by  $p$  by filling in missing details”. This statement is also executed synchronously by the group of negotiators of the plan referred to by  $p$ . By jointly executing Commit-statements, the negotiators end the plan creation phase of this plan. This doesn’t have to mean that all plan features have been totally determined by the negotiations (CommGroups) of these agents, but just that the negotiators have established all their demands on the plan and that they don’t care about the further details. Now it’s up to the constraint solver to fill in the rest of the plan. We assume the constraint solver incorporates planning knowledge in its associated *planning function*  $\varpi : \mathcal{L} \rightarrow \mathcal{L}$  to allow it to finish plans. This function takes a constraint on a plan, which describes a set of possible plans, and returns a strengthened, definite constraint which only allows one plan, which means that the new constraint prescribes one specific value for each of the six plan features. We demand that  $\varpi(\varphi) \models_{\text{cs}} \varphi$ , that is, the resulting constraint on the plan must imply the original constraint. This way, the demands of the *Negotiators* always are respected. The planning knowledge can be very powerful, enabling the constraint solver to even construct plans when the agents didn’t add any constraints to the plan store. The constraints of the *Negotiators* reduce the size of the search space for the constraint solver, which speeds up the planning process. A Commit-statement fails in case of absence of one or more negotiators in the synchronous execution. This is the semi-formal semantics:

Whenever a number of agents  $J \subseteq \mathcal{I}$  try to commit to a plan by each ( $\iota \in J$ ) executing  $\text{Commit}(p_\iota)$ , then this will succeed if all  $p_\iota, \iota \in J$  are bound to the same plan name  $P$  and  $\mu(P) \models_{\text{cs}} \text{Negotiators} = J$  (that is, the proper group of agents present). If successful, the operation updates the plan store of  $P$ , such that the resulting store is definite. Formally:  $\mu(P) := \varpi(\mu(P))$ . Finally, the Commit statement adds the plan name  $P$  to the set  $\rho$  of plans which are ready to be executed.

Note that this rule doesn’t yield a boolean indicating whether the Commit operation succeeded. This is so because when all *Negotiators* take part in the Commit operation by synchronously executing the Commit statement, the operation always succeeds. If not all *Negotiators* are present, then the global transition can’t be taken, so the agents wait until all *Negotiators* Commit to the plan. As the planning function  $\varpi$  yields a definite plan, it is safe to add the plan to the set  $\rho$  of definite plans which are ready to be executed.

Assuming that  $J \subseteq \mathcal{I}$  is a set of agent names,  $P \in \mathcal{PN}$  is a plan name,  $A_\iota = \langle S_\iota, \pi_\iota \rangle$  and  $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$  this is the global transition rule for the Commit statement:

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{Commit}(P)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi, \rho \cup \{P\}, \mu' \rangle}$$

where the following conditions hold:

- \*  $\mu(P) \models_{\text{cs}} \text{Negotiators} = J$
- \*  $\mu' = \mu[\varpi(\mu(P))/P]$

After committing to a plan, the plan creation phase for this plan has ended and the plan execution can start. To start the execution phase, we have  $\text{Execute}(p)$ , which simply means “Execute the plan referred to by  $p$ ”. This statement again has to be synchronously performed by the negotiators of the plan, and it results in starting a plan controller for the plan referred to by  $p$ . A plan controller is a special process that coordinates and synchronises plan execution and takes care of proper execution of the plan. The global semantics of an Execute statement transfers the plan name from the set  $\rho$  (plans ready to be executed) to the set  $\chi$  (plans in execution). As the global transition rule is relatively simple, we skip the semi-formal semantics and present the global transition rule. Again, assume that  $J \subseteq \mathcal{I}$  is a set of agent names,  $P \in \mathcal{PN}$  is a plan name,  $A_\iota = \langle S_\iota, \pi_\iota \rangle$  and  $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$ .

$$\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{Execute}(P)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi \cup \{P\}, \rho \setminus \{P\}, \mu \rangle}$$

where the following condition holds:

$$\mu(P) \models_{\text{cs}} \text{Negotiators} = J$$

This rule only implements the *start* of the plan execution, in which the plan is transferred to the set of plans in execution. The plan controller which is associated with the plan then directs the agents that are *Actors* of the plan to execute the actions of the plan. We will restrict ourselves to an informal account of plan execution, because plan execution takes place at the lower agent level. An execution step of a plan controller amounts to finding the minimal elements of the partial order of the plan, and executing these, either simultaneously or one after the other. Execution of an action  $a$  of the plan means synchronising

this action with the agent(s) involved, as laid down in *Actors(a)*; these agents take care of the actual execution. When the action has been done, it is removed from the set of actions of the plan. In case the action to be executed is a  $!\psi$ , then the goal  $\psi$  is broadcast in the hope there will be a group of agents willing to form and execute a plan resulting in  $\psi$ . If so, a new plan–execute cycle takes place and the controller of the outer plan suspends until such a plan has been made and executed. The plan made in this inner plan–execute cycle can contain other see–to–it statements, recursively resulting in new cycles. In case the plan fails during execution, for example because there are no agents willing to make a plan for a goal which has been broadcast through a see–to–it action, then the plan controller informs the negotiators of the plan, such that they can replan. The advantage of using plan controllers for plan execution over letting the agents execute the plan on their own is that it minimises communication between the actors. An example of a plan part that can be easily executed using a coordinator is  $a < b$  (no actions in between), with  $Actors(a) = \{\iota_1, \iota_2\}$  and  $Actors(b) = \{\iota_3, \iota_4\}$ . So, two consecutive group actions have to be performed by disjoint groups of agents. Without a plan controller, this would involve communication between these two groups.

As explained in the previous paragraph, the start of the execution phase is decided by the *Negotiators* of the plan, not by the *Actors*. We now have to direct our attention towards the group of actors. Until this point, we simply assumed that this group of actors will accept the plan and the role delegation made by the planners. As agents are (supposed to be) autonomous, we can only assume this if all actors were part of the group of negotiators, or if all actors stand in an authority relation below at least one planner. Nothing in our coordination architecture enforces this, so it is possible that an actor is suddenly called upon by the plan controller to perform an action, while this actor agent is otherwise engaged and not willing to assist in the plan at all. It can be argued that this aspect of the architecture conflicts with agent autonomy. This problem is easily solved; we could add a coordination statement to the language to inform actors which aren't negotiators about the contents of the plan. Then, we change the semantics of the Execute-statement by demanding that both the negotiators and the actors synchronously execute it. This way, the actors can refuse their cooperation. If they agree to take part in the plan execution, a (joint) commitment towards the plan can be established among the acting agents.

The reader might wonder why we opted for separate groups of negotiators and actors, which can overlap, coincide, or even be disjoint. The reason for this is that we gain expressivity. Our coordination language allows 'manager agents' to construct plans through negotiation, which 'labourer agents' have

to execute. The language also allows the negotiators to be part of the group of actors, or vice versa. These possibilities yield a coordination language which is suitable for writing coordination protocols for different organisational profiles.

The global semantics which we defined in this section gives a high-level description of the execution of the coordination statements. We abstract from implementation issues and from the mental consequences of executing coordination statements. Here, we will make some brief remarks on these matters. The statements of the coordination language need to be implemented in a lower-level language, such as an agent programming language or JAVA. A synchronous coordination statement performed by a group of agents is translated to a coordination protocol in the lower-level language, in which the agents send messages to each other, check their mental state to find out whether they are willing to take part in the coordination, and perhaps establish mutual beliefs or joint intentions. The coordination protocol underlying a coordination statement can be fairly complicated, and it is one of the benefits of using a coordination language that we can abstract from these hairy details.

Depending on the mental make-up of the agents, part of the coordination protocols which implement coordination statements might be concerned with establishing (joint) mental attitudes [83]. For example, when agents form a group, they can establish a joint goal towards the purpose of the plan. During the following group communication process the agents execute several CommGroup statements, to establish their demands on the plan being formed. In between the execution of CommGroups, individual agents can use their individual beliefs and goals to decide which constraints they want to post. When the agents Commit to the plan, this could result in a joint intention to perform the actions of the plan (in case the actors all are negotiators). And when the plan is executed, the agents could first check whether there is mutual belief in the presence of the *Circumstances* in which the plan can be executed, and when the plan is finished, establish mutual belief that the *Purpose* of the plan has been achieved.

## 6.4 Illustration

We give an extensive example to illustrate the use and usefulness of the coordination language and the functioning of the plan–execute cycles. In this example we use **this font** for agent names and actions, and *this font* for logical formulas, like constraints and goals of plans. The domain of the example is imaginary (we picture a large group of robotic agents devoted to running the household of the author of this thesis), but serves well to illustrate the various subtleties of

agent coordination.

EXAMPLE 6.3 (Cleaning up the living-room)

Agent James wants the living-room in the house of his owner to be meticulously clean. This isn't an easy job, as his owner is a sloppy woman, who isn't very into cleaning. So, James reckons he could do with some help. The household is run by five agents, called Martha, Paul, Mary, John and James. Each of them is programmed in an (agent) programming language. Different agents could be programmed in different programming languages, but they all use the coordination language defined in the previous section to coordinate plan formation and execution. Thus, the programs of the agents are interspersed with coordination statements.

James starts by trying to form a group to achieve the condition *room\_clean*. The circumstances before the cleaning activities of the agents are started is *room\_messy\_and\_dirty*. There are dirty dishes on the table, and there are old newspapers and empty potato chips bags lying around. James wants Martha to be part of the group of negotiators concerned with the plan to be formed, as she is very good in managing large cleaning projects. Martha also agrees that a plan needs to be formed to clean the room, so she also executes a FormGroup action. She demands that agent John is not part of the negotiation group, for reasons that will become apparent later on. But she wants somebody else to aid in the planning; she wants the group of negotiators to have at least two members. John is trying to have a say in the plan, so he is also trying to execute a FormGroup-statement. These are the coordination statements James, Martha and John attempt to execute:

James: FormGroup(*room\_messy\_and\_dirty*, *room\_clean*,  $p_1$ ,  
Martha  $\in$  Negotiators)

Martha: FormGroup(*room\_messy\_and\_dirty*, *room\_clean*,  $q_{18}$ ,  
John  $\notin$  Negotiators  $\wedge$  #Negotiators  $\geq 2$ )

John: FormGroup(*room\_messy\_and\_dirty*, *room\_clean*,  $r_3$ ,  
#Negotiators  $\geq 3$ )

All three agents use a different plan variable to refer to the plan they want to create ( $p_1$ ,  $q_{18}$  and  $r_3$ , respectively). The system will choose a plan name for the plan and associate this name with the variables of the agents. If the agents were to choose the plan name, then they each had to know the names of all plans that have been created in all corners of the multi-agent system. Each agent uses plan variables from its own, disjoint set (for example, all plan variables of James start with  $p$ ). If all agents would use the same set of plan variables, then name clashes could occur. For example, then agent

James could use the variable  $q_{18}$  to refer to a plan to arrange a surprise party for his owner. If the system then tries to associate  $q_{18}$  to the cleaning plan, this creates a problem for James, as the variable  $q_{18}$  already denotes another plan of his.

The above combination of FormGroup-statements can't succeed, as the group {James, Martha, John} doesn't satisfy the demands of the three agents. To explain the semantics of the coordination language, we look at the functioning of the global transition rule for FormGroup in this particular case. There are three agents that execute FormGroup statements, which can in principle be synchronously executed, if the conditions of the transition rule are satisfied. We first take  $J$  (the set of synchronising agents) to be {James, Martha, John}. For each of these three agents there is a local transition labelled with the FormGroup statement which that agent tries to execute, as shown above. For example, the local transition of James will look

like  $A_{\text{James}}$   $\xrightarrow{\text{FormGroup}(\text{room\_messy\_and\_dirty, room\_clean, } p_1, \text{Martha} \in \text{Negotiators})}$   $A'_{\text{James}}$ .

The transition rule stipulates that the first and second arguments of the FormGroups of the three must be the same, and this is indeed the case. We now look at the conditions of the transition rule. The second condition is the essential one here; it states  $\bigwedge_{i \in J} \varphi_i \wedge \text{Negotiators} = J \not\models_{\text{cs}} \perp$ . When we instantiate this formula for this particular case, we get

$$\text{Martha} \in \text{Negotiators} \wedge \text{John} \notin \text{Negotiators} \wedge \#\text{Negotiators} \geq 2 \wedge \#\text{Negotiators} \geq 3 \wedge \text{Negotiators} = \{\text{James, Martha, John}\} \not\models_{\text{cs}} \perp.$$

This condition isn't satisfied (because of John), so no global transition results from the FormGroup statements of the three agents.

But there is exactly one subset that can succeed, and this consists of the statements of Martha and James. The group {Martha, James} is the only subset for which the second condition of the global transition rule is satisfied, as can be easily checked. They become the *Negotiators* of a plan the system calls PlanA. The first condition of the transition rule stipulates that the plan name PlanA must be previously unused. The name PlanA is part of the right hand side of the resulting global transition, such that the name is available to the local programs of James and Martha to bind to their plan variables. The result of the FormGroup is that the *Purpose*, the *Circumstances* and the group of *Negotiators* of PlanA are fixed;  $\text{Purpose} = \text{room\_clean} \wedge \text{Circumstances} = \text{room\_messy\_and\_dirty} \wedge \text{Negotiators} = \{\text{James, Martha}\}$  is the current constraint on PlanA, as the third condition stipulates.

So now, Martha and James can start planning. They use their own

knowledge and reasoning capacities to decide which actions can contribute to the goal *room\_clean*. Note that the agents can have different ideas about what this goal incorporates. Though our coordination architecture doesn't focus on internal agent processes, we look inside the knowledge bases of the agents for a moment to illustrate this. Martha believes the formula  $room\_clean \leftrightarrow no\_dust\_on\_objects \wedge ashtray\_empty \wedge fresh\_atmosphere$ , while James thinks  $room\_clean \leftrightarrow no\_dust\_on\_objects \wedge fresh\_atmosphere \wedge ceiling\_clean \wedge floor\_very\_clean$ . The knowledge of the agents about details of the goal is partial. This leads to different decisions about the actions to be done, which complement each other:

James:  $CommGroup(\{!floor\_very\_clean, dust\_left\_side, dust\_right\_side\} \subseteq Actions, p_1)$

Martha:  $CommGroup(empty\_ashtray \in Actions \wedge Actors(empty\_ashtray) = \{John\} \wedge open\_windows \in Actions, q_{18})$

Both agents want to add constraints on the plan feature *Actions*, and Martha also has a constraint on the feature *Actors*. The agents want to add both atomic actions and see-to-it actions to the plan. As the constraints of the agents are consistent, and they are both present to synchronously execute a *CommGroup*-action, this succeeds. We again look at the functioning of the global transition rules for *CommGroup* in some detail. The group  $J$  here is  $\{James, Martha\}$ . The plan variables  $p_1$  and  $q_{18}$  have been locally bound to *PlanA* in the *FormGroup* which initiated the plan creation phase, so we take  $P$  to be *PlanA*. There are two local transitions for the *CommGroups* of the two agents, each labelled with the *CommGroup* statement of that agent. We assume the local semantics takes care of replacing the plan variables the agents use as second parameter with the plan name to which they are bound, which is *PlanA* here. Thus, the second parameter of both *CommGroup* statements are the same, as the antecedent of the transition rule demands. We now look at the conditions attached to the rule. The first condition is  $\mu(PlanA) \models_{cs} Negotiators = \{James, Martha\}$ , and this indeed is the case, as is clear when looking at the constraint  $\mu(PlanA)$  that resulted from the formation of the group; this is  $Purpose = room\_clean \wedge Circumstances = room\_messy\_and\_dirty \wedge Negotiators = \{James, Martha\}$ . The (instantiated) second condition states that the potential new constraint must be consistent:

$$\begin{aligned} & \mu(\text{PlanA}) \wedge \\ & \{!floor\_very\_clean, dust\_left\_side, dust\_right\_side\} \subseteq \text{Actions} \wedge \\ & \text{empty\_ashtray} \in \text{Actions} \wedge \text{Actors}(\text{empty\_ashtray}) = \{\text{John}\} \wedge \\ & \text{open\_windows} \in \text{Actions} \quad \not\models_{cs} \perp. \end{aligned}$$

This condition is satisfied, and therefore the rule for *successful* CommGroup is used. In case this condition wasn't satisfied, the transition rule for a *failing* CommGroup would have led to a global transition. The third condition takes care of the updating of the plan store with the new constraint.

As a result of this action, the plan now contains a number of unordered actions, one of which (`empty_ashtray`) is allocated to John (who hates this chore (which is the reason he tried to join the *Negotiators*. . . ; Martha always does this!)). In the next coordination action, the agents impose some ordering and allocation constraints upon the actions. Martha thinks that first the windows have to be opened, unless it is part of the plan to wash the windows. So, she first tests the plan, as James could have added the `wash_windows` actions to the plan:

Martha:  $?(wash\_windows \in \text{Actions}, q_{18})$

This test fails, so Martha can demand that the `open_windows`-action precedes all others. James also tests the plan, and finds out that `open_windows` is part of PlanA. He then tries to allocate this job to himself, and to prevent that Mary has to dust the left side of the room by herself:

Martha:  $\text{CommGroup}(\forall a \in \text{Actions} : \text{open\_windows} \leq a \wedge$   
 $\text{Actors}(\text{open\_windows}) = \{\text{Mary}\}, q_{18})$

James:  $\text{CommGroup}(\text{Actors}(\text{open\_windows}) = \{\text{James}\} \wedge$   
 $\text{Actors}(\text{dust\_left\_side}) \neq \{\text{Mary}\}, p_1)$

This CommGroup fails, as the agents have a conflict on the task allocation of `open_windows`. The failing of the group communication doesn't mean that execution blocks, but that the second transition rule for CommGroup is used instead of the first. The first condition of this rule stipulates that the proper group of *Negotiators* has to be present, as is the case. The second condition states that the conjunction of the current plan constraint with the new proposals of the *Negotiators* leads to  $\perp$ . This condition also holds, because of the conflict on the task allocation. Thus, a global transition is generated in which the local states of Martha and James stay the same, except for their programs, which advance to the statements following their respective CommGroups.

Both agents try again, relaxing or changing their constraints:

Martha:  $\text{CommGroup}(\forall a \in \text{Actions} : \text{open\_windows} \leq a, q_{18})$

James:  $\text{CommGroup}(\text{Actors}(\text{open\_windows}) = \{\text{Paul}\} \wedge$   
 $\text{Actors}(\text{dust\_left\_side}) \neq \{\text{Mary}\}, p_1)$

This does succeed, and the constraints are added to the plan store of PlanA. Now, James checks whether the plan incorporates cleaning the ceiling, and tries to add this action if not:

James:  $?(clean\_ceiling \notin \text{Actions}, p_1);$   
 $\text{CommGroup}(clean\_ceiling \in \text{Actions}, p_1)$

Note that we use ‘;’ for sequential composition, so we assume that the lower level programming language in which James is programmed contains this. The test succeeds, and the  $\text{CommGroup}$  is attempted, but as Martha is not responding by executing a  $\text{CommGroup}$ , the coordination fails and the cleaning of the ceiling isn’t added to the plan. This failure is of the blocking kind; no global transition is generated for the group communication, and James either has to wait until Martha responds and communicates with him, or James’s program can continue with another branch (if the local agent programming language contains some choice constructor).

Following this, both agents decide that they have put enough constraints on the plan, and they commit to it:

Martha:  $\text{Commit}(q_{18})$

James:  $\text{Commit}(p_1)$

When this is executed, the constraint solver of the coordination architecture takes the plan and adds enough constraints to it to result in a definite plan that attains the purpose specified by the negotiators. In this case, the plan needs more actions ( $\neg \text{room\_messy}$ ), actions need to be ordered (first, the mess must go, then the room has to be dusted, and then the floor has to be cleaned) and tasks have to be allocated. Figure 6.2 shows the resulting plan.

Now, the plan can be executed by the actors. Martha and James start execution by synchronously performing an  $\text{Execute}$ -statement, and a plan controller for PlanA is started. The plan controller starts by telling Paul to open the window. Fresh air and light stream into the living-room, thereby revealing a pile of laundry in a dark corner. After this action is done, there are two actions in the plan which have not been ordered by the planning of the agents or of the constraint solver. So, the plan controller can decide to order them arbitrarily, or execute them in parallel. We suppose the plan controller opts for concurrency. John empties the ashtray, and at the

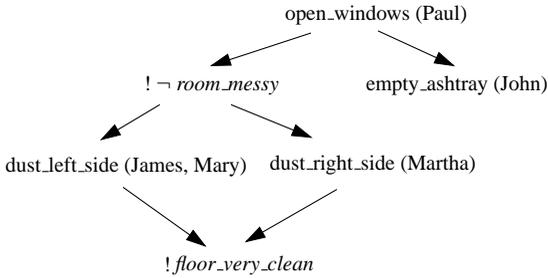


Figure 6.2: Plan A

same time the plan controller broadcasts that a plan needs to be created and executed for the purpose  $\neg room\_messy$ , after which the plan controller suspends until this new plan is formed and executed. Note that it proves useful to postpone the planning for  $\neg room\_messy$ , as the opening of the windows has revealed more mess to be thrown out. If a plan had been made *before* the windows had been opened, than this plan wouldn't have included actions for putting the laundry away.

Three agents react to the broadcast message, by doing a FormGroup-statement:

John:  $FormGroup(room\_messy, \neg room\_messy, r_8, Mary \in Negotiators)$

Mary:  $FormGroup(room\_messy, \neg room\_messy, s_2, John \in Negotiators)$

James:  $FormGroup(room\_messy, \neg room\_messy, p_2, \#Negotiators < 4)$

The group is formed with no problems, the system calls the plan PlanB and the negotiations on the plan to clean out the mess begin. John and Mary would love to do chores, as long as they can do them together (they have hidden motives...).

John:  $CommGroup(put\_laundry\_in\_washing\_machine \in Actions \wedge$   
 $Actors(put\_laundry\_in\_washing\_machine) =$   
 $\{Mary, John\}, r_8)$

Mary:  $CommGroup(remove\_garbage \in Actions \wedge$   
 $Actors(remove\_garbage) = \{Mary, John\}, s_2)$

James:  $CommGroup(remove\_garbage \in Actions \wedge$   
 $Actors(remove\_garbage) = \{James\}, p_2)$

As the agents don't agree on the task allocation, this negotiation step fails. The agents try again, where James weakens his demand, and the others

change theirs. James now wants one agent, not busy with other tasks in this plan, to remove the garbage. Mary and John also change tactics; they try to get the other agents out of the room by allocating tasks to them, such that they end up in the living-room together:

John:  $\text{CommGroup}(\text{put\_laundry\_in\_washing\_machine} \in \text{Actions} \wedge$   
 $\text{Actors}(\text{put\_laundry\_in\_washing\_machine}) =$   
 $\{\text{Martha}\}, r_8)$

Mary:  $\text{CommGroup}(\text{remove\_garbage} \in \text{Actions} \wedge$   
 $\text{Actors}(\text{remove\_garbage}) = \{\text{James}\} \wedge$   
 $\text{remove\_dishes} \in \text{Actions} \wedge$   
 $\text{Actors}(\text{remove\_dishes}) = \{\text{Paul}\}, s_2)$

James:  $\text{CommGroup}(\text{remove\_garbage} \in \text{Actions} \wedge$   
 $\exists x : \text{Actors}(\text{remove\_garbage}) = \{x\} \wedge$   
 $\forall a : a \in \text{Actions} \wedge a \neq \text{remove\_garbage} \rightarrow$   
 $x \notin \text{Actors}(a), p_2)$

The constraints of the agents now are consistent, and so the three actions are added to PlanB. All three actions are allocated to different single agents, as the constraints prescribe. These agents don't have to be part of the *Negotiators*-group. Note that it seems that Mary and John have communicated about their change in tactics in order to obtain what they want. If they did, this is lower-level communication, performed using communication statements from the lower-level programming language. The higher-level statements from the coordination language are used to add well-considered proposals of the negotiators to the plan. In case the agents haven't considered their proposals in enough detail, for example because they didn't find out whether other negotiators agree with their opinions on the plan in formation, high-level communication fails.

The three agents Commit to PlanB, and because the three actions of the plan can achieve its purpose and there is no need to order the independent actions, the constraint solver returns the plan with the three unordered actions. Then, the negotiators initiate execution of the plan, after which the agents Martha, James and Paul execute this plan. A new plan controller starts, to coordinate the execution, and after the plan is finished, the plan controller of PlanA takes over again. The plan of Figure 6.2 is continued. The room is dusted concurrently by Martha, James and Mary. After this is finished, the plan controller again has to execute a see-to-it-action, namely *!floor\_very\_clean*. Again, this goal is broadcast and the plan controller of PlanA suspends.

Martha and Mary react to the broadcast, and they successfully form a group to create a plan for cleaning the floor. In order to thoroughly clean the floor, the big couch has to be lifted such that an agent can vacuum the floor underneath it. Lifting the couch has to be done by at least two agents. During the first CommGroup-action, Martha proposes a set of actions, while Mary agrees with anything:

Martha:  $\text{CommGroup}(\text{Actions} = \{\text{lift\_couch}, \text{vacuum\_under\_couch}, \text{put\_couch\_back}, \text{vacuum\_rest}\} \wedge \#Actors(\text{lift\_couch}) = 2, q_{21})$

Mary:  $\text{CommGroup}(\top, s_4)$

After this, Mary performs a series of tests on the plan store of this plan (called PlanC), to find out whether lift\_couch is part of the plan (this succeeds), whether lift\_couch has already been allocated (this fails), and how many agents are needed to lift the couch. We will show the last two test statements.

The test  $?(\text{Actors}(\text{lift\_couch}) = x, s_4)$  uses the free local variable  $x$ . In case the constraints on PlanC imply that there is a group of agents that will lift the couch, this set will be bound to  $x$ . As the constraints presently don't imply this, this test fails.

The test  $?(\#(\text{Actors}(\text{lift\_couch})) = y, s_4)$  does succeed, and the value 2 is bound to the local variable  $y$ . Now that Mary knows that only two agents will have to lift the couch, she proposes a new plan constraint. Meanwhile, Martha puts some ordering constraints on the actions:

Martha:  $\text{CommGroup}(\text{lift\_couch} < \text{vacuum\_under\_couch} < \text{put\_couch\_back}, q_{21})$

Mary:  $\text{CommGroup}(\forall x \in \text{Actors}(\text{lift\_couch}) : \text{strong}(x), s_4)$

These coordination statements succeed, and subsequently Martha and Mary commit to the plan. As the constraint solver contains the domain knowledge that the only strong agents are John and Paul, it allocates the group action of lifting the couch to them. The constraint solver allocates the two vacuuming actions to Mary, and adds an ordering constraint to first vacuum the rest of the room before vacuuming under the couch. The agents execute PlanC, after which the overall plan PlanA also terminates.

## 6.5 The coordination architecture

In Figure 6.3, we depict the coordination architecture described earlier.

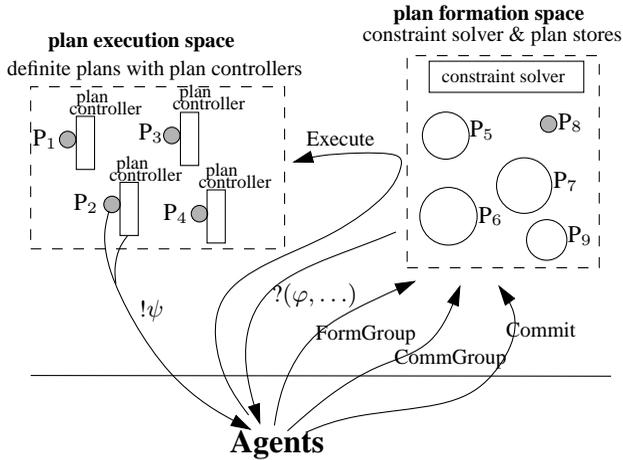


Figure 6.3: The distributed plan coordination architecture

There are two levels in the architecture. In the lower level are the agents, programmed in some (agent) programming language. The ‘ordinary’ statements of the agent programs are executed in the lower level. The higher level is the plan coordination level, on which we focus in this chapter. This level contains two spaces: the plan formation space and the plan execution space. The plan formation space contains the plan stores of plans still in the plan creation phase. In the figure, each store is indicated by a circle. The size of the circle indicates the size of the set of plans which are consistent with the constraints in the store; that is, the smaller the circle, the further the creation of the plan has advanced. The little grey circles indicate plan stores which are definite and thus only allow one plan. These plan stores can be equated with plans. The constraint solver interacts with the plan stores, checks the constraints the negotiating agents want to add to plan stores and uses its planning knowledge to aid in creating and finishing the plan. The plan execution space contains finished plans, which are being executed. Each plan has an associated plan controller to coordinate plan execution with the agents involved.

Looking at the global system configuration  $\langle \{A_i | i \in \mathcal{I}\}, \chi, \rho, \mu \rangle$ , we can pinpoint the spaces of the architecture:  $\{A_i | i \in \mathcal{I}\}$  are the agents in the lower level,  $\chi$  are the plans in the plan execution space,  $\rho$  matches the plans that are definite (the little grey circles from Figure 6.3), and  $\mu$  are the other plans in the plan formation space.

As the figure shows, there can be many plans under construction and in exe-

cution simultaneously. This feature of the architecture is useful, as independent plan coordination activities can be done in parallel. One group of agents doesn't have to wait until another group has finished executing its plan before starting on a new plan. Of course, multiple plans of possibly overlapping groups of agents require more deliberation and communication of the agents, to decide whether they are available to contribute to a plan. This reasoning takes place at the lower agent level.

The programs of the agents contain statements from the plan coordination language, which are executed at the higher level. The arrows in the figure mainly indicate data flow. For example, when agents do a `CommGroup`, information is added to a certain plan store. When agents test a plan store, agents obtain information about the plan being formed. Execution of coordination statements in the plan formation store involves the constraint solver. For example, when a group of negotiators commits to a plan, then the constraint solver checks whether this plan is definite, and if not, uses its planning knowledge to arrive at a definite plan. When a finished plan is executed, it moves from the plan formation space to the plan execution space and gets a plan controller associated. The plan controller coordinates the agents to perform the actions of the plan, according to the order dictated by the partial order. So, if the next action to be executed is a group action for agents  $\{\iota_2, \iota_5, \iota_6\}$ , the plan controller synchronises with these agents, such that they perform the action together. When the next action to be executed is a `see-to-it` action, the plan controller broadcasts this to the agents, such that a new distributed plan-execute cycle can start, inside the current one. After execution of the plan is finished, the plan is disposed of. In case a plan fails during execution, for example because not all actors are present to execute a group action or because no agent reacts to a `see-to-it` message, the plan controller reports the failure to the negotiators of the plan, after which it terminates.

## 6.6 Conclusion

We defined an architecture for agents that coordinate with each other in order to create and execute plans. Also, we introduced a novel coordination language which can be used to enable agents to form temporary alliances for planning and performing the group plan. Coordination languages are very useful for agent applications, as agents often are heterogeneous and thus need a common language of a high abstraction level to coordinate their behaviour. This coordination language uses ideas from constraint programming [111] and is similar to GrAPL [122], the language to coordinate group actions from Chapter 5. When

we compare the coordination language to GrAPL, GrAPL is more expressive with respect to details of the actions: actions are parameterised and the agents negotiate about the parameters. A combination of GrAPL and the coordination language is well conceivable. In this new language, there are constraints on two levels, namely the constraints on the structure of the plan and the task allocation, and the constraints on parameters of actions which are part of the plan.

Our work is also related to classical Linda-like coordination languages [17], which use a blackboard to exchange information. Instead of the blackboard, our architecture features constraint stores, which have the added benefit of logical structure and inference mechanisms.

In a way, the coordination language can be compared with agent communication languages like KQML [46]. The coordination statements (FormGroup, CommGroup, Commit and Execute) correspond to speech act types, and the constraints and other parameters of these statements form the content of the messages. The statements of our language are of a higher abstraction level; one can imagine implementing them in KQML. The benefit of our language is that it is specifically designed for the coordination of agents creating and executing plans, and that its semantics precisely describes the manipulation of the plan stores. In order for communication and coordination to be successful, the agents must have ontological agreement on the language they use. It remains future work to show how our coordination language can be useful in negotiating about ontological issues.

The coordination architecture is generic, and is to be combined with planning methods for the agents and the constraint solver. The agents use these planning algorithms to decide which constraints they want to add to a plan store, and the constraint solver uses a planning algorithm to check whether the combined proposed constraints of agents indeed contribute to a plan that reaches its purpose, as well as to complete the plan when the agents have committed to it. By abstracting from specific planning methods, the coordination language can be suitably used by agents using different planning approaches.

Plans in our approach are partially ordered sets of actions. This matches with the plan format in partial order planning (eg., [97]), where plans are formed by gradually constraining the action set and the order of the actions, as is necessary to obtain the goals of the plan. Planning algorithms like these can be very well combined with constraint satisfaction techniques, to be implemented in the constraint solver of our coordination architecture. Other, related planning methods which combine well with our coordination architecture are distributed hierarchical task network planners [27], which work by gradually refining a goal into subgoals and/or actions. Recently, a new generation of fast

planners like Graphplan [11] and satisfiability based planning [78] has arisen. These planners are not of the usual deductive kind, but use satisfiability instead. As this view matches with constraint satisfaction, these new planners are very usable in the coordination architecture. There are also planners that formalise the planning problem as a constraint satisfaction problem, and then use constraint solving to achieve a solution [6]. This results in a very fast planner, according to the authors of [6]. In this planning method, the variables which are constrained describe the *states* which are achieved by performing actions, while in our formalism the variables (plan features) describe the *actions* in the plan and their properties. The multi-agent constraint-based planner MACBeth [53] combines constraint programming with hierarchical task network planning. The (human) user and the planner cooperate to tailor a sketchy plan to a specific situation. The user posts constraints on the relations between sub-tasks in the plan under construction, and on the value of task parameters. Thus, the constraints both describe demands on the structure of the plan and on the resource allocation. MACBeth uses constraint management techniques to arrive at a definite plan. This seems similar to the set-up in our coordination architecture, where the agents which negotiate about the plan first post constraints on the plan, thus narrowing down the search space for the constraint solver. But our coordination architecture is of a different nature; it isn't entered around a planner (while MACBeth is), but provides a generic system structure for agents doing distributed planning and a language through which the agents can jointly post their demands on the plan.

In this chapter, we only briefly touched upon mentalistic aspects of group activity [33, 103, 118]. We see work in this area as valuable, and establishing connections between the construction of plans and the proper mental attitudes as very relevant. But presently, we chose to focus on plan construction and execution. Another relevant field of research concerns the structuring of coordination processes, for example using coordination protocols [95]. Here, we don't mean the lower-lever coordination protocols used to implement the coordination statements, but the higher-level coordination protocols constructed using statements from the coordination language. Structuring coordination is essential, to guide the agents in using the proper coordination statements at the right times. This prevents them from arbitrarily trying to achieve agreement on matters, which will take a lot of communication effort. Our coordination language can be used to implement coordination protocols.

Our coordination architecture and language enable the agents to negotiate about demands on the plan being created by posing constraints on the plan. These constraints shrink the space of possible plans, after which the planner implemented in the constraint solver can complete the plan. This way, the

preferences of the agents and the knowledge of the planner can be combined. The coordination language offers abstract and formally well-defined primitives, enabling agents to interact through the construction and execution of distributed plans.



# CHAPTER 7

---

## **Conclusion: Abstract Interaction is what Agents are for!**

---

*What you don't have you don't need it now  
What you don't know you can feel somehow*

*U2*

As we studied several entirely separate aspects of agent interaction and applications of abstraction for multi-agent systems, it is hard to come to all encompassing conclusions at the end of this thesis without resorting to rather empty platitudes. Therefore, we sweep together the main conclusions of the chapters in this book in an overview. Then, we briefly touch upon some directions for future work. And as platitudes have their own particular charm, we will not resist temptation and conclude with some more general remarks.

### **Summary of chapter conclusions**

#### **Chapter 2:** *Abstracting Agent Interaction Histories into Intentionality*

We have defined formal criteria which relate intentional notions to histories of an agent's interaction with its environment. To be specific, we

relate beliefs, desires, and intentions of an agent to the observations and communication an agent receives and the actions it performs over time. From an external viewpoint, an agent is a black box, with its behaviour history (past actions and received observations and communications) as input, and its new actions as output. Like we do with our fellow human beings, it can be very useful for an agent to explain the behaviour of a fellow or rival agent in terms of belief, desire and intention, in order to be able to understand and predict the actions of this other agent. Thus, the intentional notions serve to *abstract* the *interaction histories* of the agent into manageable, short representations.

The criteria for being an externally grounded belief, desire or intention formula formally relate the abstract, intuitive BDI notions to the computational reality of traces of the agent's observable behaviour. As we provide formal definitions of BDI in terms of dynamic behaviour, our work in a way formalises the philosophical account of Dennett [25, 26], promoting the intentional stance. But the definitions also work the other way around, by giving the usually rather vague BDI notions a rigorously defined meaning.

The criteria for intentional notions which we have defined can be used in several ways:

- \* Agents can use the criteria to attribute intentional notions to another agent and to perform anticipatory reasoning about future behaviour of this agent, which may lead to attempts to make the other agent change its mind.
- \* In case agents internally use BDI-notions, the criteria for BDI can be used to verify whether belief, desire and intention have the proper relation to the external agent behaviour, as fixed in the criteria.
- \* The abstract BDI-notions can be helpful for requirements engineering, which is concerned with specifying requirements that systems have to live up to. These requirements often concern complex behaviour, and our BDI-notions can be used to abstract behaviour patterns into shorter, more intuitive concepts, which still have a well-defined relation to the behaviour of the system. This aids in managing the complexity of the requirements.

### Chapter 3: Reuse and Abstraction in Verification: Agents Interacting with Dynamic Environments

In this chapter, we have provided two principles to make verification of agent systems a more manageable process. In order to be able to verify properties, we need a specification language for properties which is closely related to the specification (or programming) language used to specify the agent system. Though there are temporal logics available that could be used to specify properties of agent systems, it is often unclear how to relate the semantics of these logics to the semantics of agent system specifications. Therefore, we started at the semantical side in this chapter.

The semantics of an agent system is assumed to be a set of traces, which represent all possible behaviours of the system. These traces are temporal sequences of information states of the agent(s) and the world. Based on the representations used in these information states, we have defined a detailed language, for phrasing behavioural properties. As this language results in rather long, unreadable, unintuitive properties, we introduced the first principle to make verification easier, which is *language abstraction*. Language abstraction means finding intuitive, conceptual terms to abbreviate temporal phrases from the detailed language, resulting in an abstract language. As each abstract term is formally related to its meaning in the detailed language, we aren't losing any formal rigour through language abstraction, and the intuitive abstract language is firmly anchored in the detailed language. Though this is an almost trivial idea, it aids considerably in reducing the inherent complexity of agent verification.

The second principle we have presented is the idea of reusable systems of generic properties and proofs. Many agent systems have behaviour in common, in an abstract sense. For example, most agents perform actions and receive observations. By *abstraction* from domain and system dependent details, we obtain abstract agent architectures, each representing a generic class of agent systems. We can phrase generic properties applying to an abstract agent architecture. These generic properties can be formalised in the abstract language, for brevity and intuitiveness. By proving the properties in a generic manner, the proofs and the properties can be reused for all agent systems which belong to the generic class of systems. All that needs to be done is instantiate domain and system dependent details and prove the underlying assumptions of the verification proofs. In this chapter, we demonstrate this principle by constructing

a system of properties and proofs for action successfulness for reactive single agents *interacting with dynamic environments*. The two principles have the following benefits:

- \* Properties in the abstract language are intuitive, and can be understood by agent software engineers without extensive training in logic.
- \* Generic properties and proofs save time and trouble, as verification results can be reused, without going into the dirty details of the verification process again.

#### **Chapter 4:** *An Abstract Model for Agents Interacting in Real-Time*

We have constructed a model for multi-agent systems which includes real-time and focuses on the *interaction* of agents with each other and with the world in which they are situated. The model *abstracts* from details of the internal mental state and of the reasoning processes of the agent. We define the model through the operational semantics of a skeleton agent programming language. The formal operational semantics unambiguously defines the meaning of the abstract programming statements of the skeleton programming language. As we don't focus on the internal make-up of the agents, the model has several parameters, such that the system builder can shape the internal agent according to his own favourite agent theory. We strictly separate internal agent processing (reasoning) and external agent processing (observing, communicating, and physical acting).

The agents execute the *reason–interact cycle*. In each round of this cycle, an agent first performs internal actions, followed by external actions. The cycle is defined syntactically, and we prove that the syntactic restrictions indeed take care of proper alternation of reasoning and interacting. As all parts of the reason–interact cycle can be completely specified in the skeleton programming language, we provide a high degree of flexibility. In other programming languages, it is fixed in the semantics of the language how incoming information is processed by the agents, and also how and when the execution of actions following from commitments of the agents takes place. (However, in [24], efforts are underway to program the cycle of the agent programming language 3APL [59]). The most important features of our agent model are:

- \* It explicitly contains a dynamic world, which is not under total control of the agents. Thus, events can take place, which could

interfere with the actions of agents. In many agent programming languages action execution is a side effect of updates to the mental state, and thus the world is not explicitly modelled. Actions can't fail in the semantical models of these languages; in our model, they can.

- \* Agents can perform observative actions, to obtain information about the world situation. If agents don't observe, they're likely to have faulty, outdated world information, leading to wrong decisions. In many other semantical models of agent programming languages, sensing is implicit and the agents always have an accurate view of the world.
- \* Physical actions have a duration, which we model by dividing each action into a number of steps or sub-actions. The semantics of the skeleton programming language doesn't interleave actions, but allows actions to take place during overlapping time frames. Thus, interference or synergy of actions can be modelled.
- \* Agents can perform actions individually or group-wise. We relate group actions to individual actions using group action schemes, which formalise the individual contributions of the group members to a group action.

As our agent model doesn't abstract away these issues, it provides a more realistic view on the interaction of agents with their environment.

### **Chapter 5:** *An Abstract Programming Language for Agents Interacting through Group Actions*

We have designed an agent programming language, called GrAPL, specifically tailored to group coordination and collaboration. Agents are social, and they can aid each other to establish their joint or selfish objectives. Facilities for negotiation about the parameters of group actions and for group action execution seem to be missing from existing agent programming languages. We define a novel constraint agent programming language with statements for exactly these purposes.

The agents can communicate their demands on the parameters of future group actions by means of constraints. The constraints of the communicating agents narrow down the parameter space of a group action, and posit demands on the composition of the group of agents which will execute the action, as well as on the group of communicators itself. The language also contains a statement for joint group action execution. This

can only succeed if the actual parameters the executing agents use obey the constraints established earlier.

The new coordination statements of GrAPL enable *high-level social agent interaction*. The new GrAPL statements for group communication and group action execution are highly *abstract*, capturing the intuitive idea that groups of agents meet in order to discuss how some future group action will be performed, and that the agents later on perform the action according to previous agreements. The formal operational semantics of GrAPL anchors the new abstract programming statements to their precise computational meaning. GrAPL is also *abstract* in the sense that the semantic model doesn't include real time or a world. Action execution is instantaneous and effects immediately affect the belief base(s) of the acting agents. Also, the language doesn't have statements for ordinary communication, observation or extensive practical reasoning. Other agent programming languages do provide facilities for these features. Summarising, GrAPL:

- \* provides abstract communication and cooperation statements, enabling groups of agents to negotiate about group actions and execute these when there is agreement.
- \* is an agent programming language with a formally defined operational semantics, which gives a rigid basis for its future implementation.
- \* uses established ideas from constraint programming, and can be implemented on top of a constraint solver; there is no reinventing the wheel.

### **Chapter 6:** *An Abstract Coordination Language for Agents Interacting in Distributed Plan-Execute Cycles*

We have defined an agent coordination language for the construction and execution of distributed plans. Agents can form temporary groups of negotiators, which communicate with each other to propose constraints on a plan to achieve a certain purpose in certain circumstances. Like GrAPL, the coordination language is constraint-based; the constraints pertain to the actions of the plan, their order, the agents executing the actions and the composition of the group of negotiators.

Being a coordination language, the language *abstracts* from all aspects not relevant to agent coordination, such as the internal reasoning of the agents. The coordination language can be used for heterogeneous agents,

programmed in different agent programming languages. It provides highly abstract coordination primitives, facilitating agent *interaction* in the form of negotiation and execution of distributed plans. The formal operational semantics of the coordination language provides a clear definition of the precise meaning of the abstract coordination statements.

We also have provided an architecture for agent coordination, which contains the agents, a constraint solver, constraint stores containing the demands of the agents on the plans and plan controllers, which coordinate the distributed execution of plans. The agents locally use planning algorithms to decide which actions, order and actors to propose. After the agents have communicated their preferences, the space of possible plans has shrunk, and the constraint solver finishes the plan, using its built-in planning knowledge. We *abstract* from the specific planning methods employed by the agents and the constraint solver; this leads to a flexible framework for distributed planning. The coordination language is:

- \* formally well-defined; it has an operational semantics, built on top of the local semantics of the programming language(s) used for the agents.
- \* constraint-based, which allows us to reuse techniques from constraint solving and constraint programming.

## Future work

Each of the separate chapters above evoke issues that need attention in future research, but there are also possibilities to integrate the work from different chapters. We start with the issues that pertain to a particular chapter.

The work in Chapter 2 (as in all of this thesis) is mainly theoretical. An implementation of an agent architecture for attribution of BDI has partly been built (see [76]). In order to make this architecture useful in practice, efficient algorithms to check the BDI criteria for candidate formulae must be included. Also, the use of the externally defined BDI notions for requirements engineering must be further investigated by analysing practical case studies.

In Chapter 3, we started a library of reusable properties and proofs for verification of agent systems by defining such a system to prove action successfulness for single agents interacting with dynamic environments. However, this is clearly just the beginning. Other systems of properties and proofs have to be constructed for this generic class as well as for other classes to arrive at an actual well-endowed library. Another matter for future research is to relate our

logical languages or subsets thereof to temporal or dynamic logics, such that we come closer to bridging the gap between implemented agent systems and agent theories. A start with this has been made in [40].

The operational agent model of Chapter 4 is an abstract model. By instantiating the parameters of the model, actual agent programming languages can be obtained. It could be interesting to mimic established agent programming languages using the operational model, by customising the skeleton programming language, and to study their differences and real-time behaviour. The model itself can also be further studied and enhanced. We just mention one major issue. The world state transformation function, which takes all atomic sub-actions and events taking place during a time unit and computes the change to the world state, is a parameter of the model. Defining this function for actual systems is by no means easy, and it might be interesting to investigate the connection between this problem and research on the frame, qualification and ramification problems. Also, theories of concurrent action could be useful to come to a definition of this function.

The main challenge for the programming language GrAPL from chapter 5 and the coordination language for distributed planning lies in implementing them. Attempts in this direction are currently undertaken.

The work in the last three chapters is clearly related, as in each of these chapters we define a (programming or coordination) language and its operational semantics. We already mentioned in the conclusion of Chapter 6 that it might be interesting to combine GrAPL and the coordination language, such that the agents constructing distributed plans can also discuss action parameters (actions are unparameterised in Chapter 6). The languages of chapter 5 and 6 both abstract from many agent aspects, such as observation, practical reasoning and asynchronous communication. Integration of the features of GrAPL and the coordination language with the features of the agent programming language for communication ACPL (by Van Eijk and others [12]) and the agent programming language for practical reasoning 3APL (by Hindriks and others [59]) might be very fruitful, and lead to a rich programming language including many useful agent features. Also, it could be quite interesting to add features from the real-time agent model of Chapter 4 to the semantical model of this resulting language, or, if this turns out to be a bit too ambitious, to the combination of GrAPL and the coordination language.

The criteria for being an intentional notion grounded in external behaviour, which we defined in Chapter 2, could be used for the verification of agents, as we performed in Chapter 3. In Chapter 3, we chose not to look inside agents; we only proved that the behaviour of the agent is such that its actions always succeed. Other properties worth verifying could concern the relation between

the BDI notions that agents use internally and their external behaviour, and the criteria for being an externally grounded intentional representation could be the properties proven in this case. Another connection worth investigating is to use the criteria for BDI in language abstraction, in order to make behaviour properties simpler and more intuitive, even if it is unknown whether the agent internally uses BDI.

The connection between the first part of this thesis (Chapters 2 and 3) and the second part (Chapters 4, 5 and 6) is less clear. The first part starts from the semantics of agent system specifications in the form of sets of traces without looking into the syntax of the specification language nor at the way the semantics results in the set of traces, while in the second part we define syntactical languages and semantical machinery, without studying the resulting set of traces. These observations also show the bridge between these two parts; by relating the operational semantics of the last chapters to the trace semantics employed in the first chapters, an integration can be established. This is by no means straightforward; as the reader may have noticed, the concepts and languages used in Chapter 2 and 3 of this thesis are different from those used in Chapter 4, 5 and 6. In order to relate both approaches, it is necessary to map the semantical models used in the last three chapters to the traces used in Chapter 2 and 3. We suspect this will take considerable effort. When this connection has been established, the verification techniques from Chapter 3 can be applied to agents programmed in one of the languages of the last chapters, and intentional notions can be attributed to them according to the criteria in Chapter 2.

## Famous last words

Three popular viewpoints on agents are agent theories, agent architectures and agent languages. (The former ATAL workshop was named after this classification.) Here, *theories* are agent logics [21, 34, 85, 102], formalising anthropomorphic concepts such as belief, desire and intentions, which are used for agent specification. *Architectures* are structural, component-wise designs and implementations of agent systems [13, 67]. *Languages* are agent programming languages [59, 101, 113], used to build agents (in the end) and to explore the necessary vocabulary to build agents (which is what they seem to be used for presently). The dependencies and connections between work from these different viewpoints often are blurred, as researchers tend to be focused on just logic or just language or just architecture. In our opinion, this cannot be a good thing. Some logical theory of agency only is useful if it can be related to implementations of agents in certain architectures or languages. And any architecture or

language needs an accompanying logic, when formal specification and/or verification is necessary. In this thesis, we have worked from all three perspectives, and this has strengthened our conviction that the agent field needs to unify the three perspectives. Having said this, we recognise that this is not easy, as we have stayed mostly within one perspective in each of the chapters of this book. Nevertheless, when the field of agent research keeps developing in divergent directions, we will never come to a stable agent concept and to multi-agent systems which will be omnipresent in everyday life.

So, what exactly is the relation between programming languages and architectures, and between programming languages and theories? It seems that the way we try to develop programming languages now, in the agent paradigm, is totally opposite to the way this happened first. Earlier programming languages started with the operations computers perform (manipulating memory places, and shifting bytes around) and then worked their way up the abstracting ladder. Theories about programs in these programming languages were constructed later, and so were architectures. We now seem to work the other way around: there already are intuitions about agents, as well as agent theories and sketches of architectures, and we try to use these in order to interpret system behaviour, define models of agents and devise programming languages. As we start from intuitions and ideas, as well as from complex agent logics, it is essential to also consider the practical aspects of agents and their complex interactions, in order to come to results which truly contribute to the future of the agent paradigm. On the other hand, one of the major benefits of the agent paradigm is that multi-agent systems start from intuitions rather than from technicalities.

We think that the work in this thesis has shown that intuitive abstractions are not just aesthetically pleasing frills, but indispensable tools to come to powerful multi-agent systems. In this book, we have captured different features of agent interaction in abstract concepts, thus making them more tangible, intuitive and usable, be it as a coordination or programming statement, as a means to interpret agent behaviour or as a phrase in a property to be verified. We have found ways to structure the enormous complexity which always comes with agent interaction into abstract concepts. We named these concepts in an intuitive manner, inspired by the anthropomorphic nature of the agent paradigm. We have always taken care of properly relating the newly found abstract concepts to the underlying complex computational reality, by either providing them with a formal operational semantics (Chapter 4, 5 and 6), defining them in terms of more detailed notions (Chapter 3) or providing criteria that relate them to execution traces of agent systems (Chapter 2). This way, we don't provide empty phrases, but instead introduce new, well-defined, powerful abstractions for agent interaction.

Agents are tailored to perform interaction with their environment. As interaction between computing components becomes ever more abundant, the agent-oriented paradigm will gain in importance and use. Equipped with the proper abstract notions for agent interaction, the agent paradigm will be even more suitable for providing the applications that are needed nowadays. Summarising the main conclusion of this thesis, agents are a powerful metaphor, offering many abstract yet intuitive concepts which contribute to the conception and construction of software entities which interact with one another and with the environment in which they exist. In short, abstract interaction is what agents are for!



---

## Bibliography

---

- [1] C.E. Alchourrón, P. Gärdenfors and D. Makinson, ‘On the Logic of Theory Change: Partial Meet Contraction and Revision Functions’, *Journal of Symbolic Logic* **50**, 1985, pp. 510–530.
- [2] J.F. Allen, ‘Maintaining Knowledge about Temporal Intervals’, *Communications of the ACM* **26**(11), 1983, pp. 832–843.
- [3] J.F. Allen, ‘Recognizing Intentions from Natural Language Utterances’, in: *Computational Models of Discourse*, M. Brady and R.C. Berwick (eds.), MIT Press, Cambridge, 1983.
- [4] J.W. de Bakker, W.-P. de Roever and G. Rozenberg (eds.), *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer, Berlin, 1989.
- [5] H. Barringer, R. Kuiper and A. Pnueli, ‘A Really Abstract Concurrent Model and its Temporal Logic’, in: *Conference Record of the 15th ACM Symposium on Principles of Programming Languages (POPL’86)*, 1986, pp. 173–183.
- [6] P. van Beek and X. Chen, ‘CPlan: A Constraint Programming Approach to Planning’, in: *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI’99)*, AAAI Press/ The MIT Press, 1999, pp. 585–590.

- [7] P. Belegirinos and M. Georgeff, 'A Model of Events and Processes', in: *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, Morgan Kaufman Publishers, Inc., 1991, pp. 506–511.
- [8] J.F.A.K. van Benthem. *The Logic of Time: A Model-Theoretic Investigation into the Varieties of Temporal Ontology and Temporal Discourse*, Reidel, Dordrecht, 1983.
- [9] M.H. Bickhard, 'Representational Content in Humans and Machines', *Journal of Experimental and Theoretical Artificial Intelligence* **5**, 1993, pp. 285–333.
- [10] S. Blamey, 'Partial Logic', in: *Handbook of Philosophical Logic, volume III*, D. Gabbay and F. Guenther (eds.), Reidel, Dordrecht, 1986, pp. 1–70.
- [11] A. Blum and M. Furst, 'Fast Planning Through Planning Graph Analysis', *Artificial Intelligence* **90**(1–2), 1997, pp. 281–300.
- [12] F.S. de Boer, R.M. van Eijk, W. van der Hoek and J.-J. Ch. Meyer, 'Failure Semantics for the Exchange of Information in Multi-Agent Systems', in: *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*, C. Palamidessi (ed.), LNCS 1877, Springer, Berlin, 2000, pp. 214–228.
- [13] M.E. Bratman, D.J. Israel and M.E. Pollack, 'Plans and Resource-Bounded Practical Reasoning', *Computational Intelligence* **4**, 1988, pp. 349–355.
- [14] F.M.T. Brazier, F. Cornelissen, R. Gustavsson, C.M. Jonker, O. Lindberg, B. Polak and J. Treur, 'Compositional Design and Verification of a Multi-Agent System for one-to-many Negotiation', in: *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS'98)*, IEEE Computer Society Press, 1998, pp. 49–56.
- [15] F.M.T. Brazier, B. Dunin-Keplicz, N.R. Jennings and J. Treur, 'DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework', *International Journal of Cooperative Information Systems* **6**, M. Huhns and M. Singh (eds.), special issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, 1997, pp. 67–94; preliminary and shorter version in: *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS'95)*.

- [16] F.M.T. Brazier, P.A.T. van Eck and J. Treur, ‘Semantic Formalisation of Emerging Dynamics of Compositional Agent Systems, in: *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 7: Agent-Based Defeasible Control in Dynamic Environments, J.-J. Ch. Meyer and J. Treur (eds.), Kluwer Academic Publishers, 2002, pp. 167–196.
- [17] N. Carriero and D. Gelernter, ‘Coordination Languages and their significance’, *Communications of the ACM* **53**(2), 1992, pp. 97–107.
- [18] K. Cetnarowicz, P. Gruer, V. Hilaire and A. Koukam, ‘A Formal Specification of M-agent Architecture’, in: *From Theory to Practice in Multi-Agent Systems, Proceedings of the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS’01)*, B. Dunin-Kępicz and E. Nawarecki (eds.), LNCS 2296, Springer, Berlin, 2002, pp. 62–72.
- [19] W.D. Christensen and C.A. Hooker, ‘Representation and the Meaning of Life’, in: *Representation in Mind: New Approaches to Mental Representation, Proceedings*, H. Clapin (ed.), University of Sydney, 2000, to be published by Springer Verlag.
- [20] A. Clark, *Being There: Putting Brain, Body and World Together Again*, MIT Press, 1997.
- [21] P.R. Cohen and H.J. Levesque, ‘Intention is Choice with Commitment’, *Artificial Intelligence* **42**(3), 1990, pp. 213–261.
- [22] D. Corkill, ‘Hierarchical Planning in a Distributed Environment’, in: *Proceedings of the 6th International Joint Conference on Artificial Intelligence (IJCAI’79)*, 1979, pp. 168–175.
- [23] A. Dardenne, A. van Lamsweerde and S. Fickas, ‘Goal-Directed Requirements Acquisition’, *Science in Computer Programming* **20**, 1993, pp. 3–50.
- [24] M. Dastani, F.S. de Boer, F. Dignum, W. van der Hoek, M. Kroese and J.-J. Ch. Meyer, ‘Programming the Deliberation Cycle of Cognitive Robots’, in: *Proceedings of the 3rd International Cognitive Robotics Workshop*, 2002.
- [25] D.C. Dennett, *The Intentional Stance*, MIT Press, Cambridge, Massachusetts, 1987.

- [26] D.C. Dennett, ‘Real Patterns’, *The Journal of Philosophy* **88**, 1991, pp. 27–51.
- [27] M. desJardins and M. Wolverson, ‘Coordinating a Distributed Planning System’, *AI Magazine* **20**(4), 1999, pp. 45–53.
- [28] F. Dignum, B. Dunin-Kępicz and R. Verbrugge, ‘Creating Collective Intention through Dialogue’, *Logic Journal of the IGPL* **9**(2), 2001, pp. 289–303.
- [29] F. Dignum, D. Kinny and L. Sonenberg, ‘Motivational Attitudes of Agents: On Desires, Obligations and Norms’, in: *From Theory to Practice in Multi-Agent Systems, Proceedings of the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS’01)*, B. Dunin-Kępicz and E. Nawarecki (eds.), LNCS 2296, Springer, Berlin, 2002, pp. 83–92.
- [30] V. Dignum, J.-J. Ch. Meyer and H. Weigand, ‘Towards an Organizational Model for Agent Societies Using Contracts’, in: *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS’02)* C. Castelfranchi and W.L. Johnson (eds.), ACM Press, 2002, pp. 694–695.
- [31] F.I. Dretske, *Explaining Behaviour: Reasons in a World of Causes*, MIT Press, Cambridge, Massachusetts, 1991.
- [32] B. Dunin-Kępicz and A. Radzikowska, ‘Nondeterministic Actions with Typical Effects: Reasoning about Scenarios’, in: *Formal Models of Agents*. J.-J. Ch. Meyer and P.-Y. Schobbens (eds.), LNAI 1760, Springer, Berlin, 1999, pp. 143–156.
- [33] B. Dunin-Kępicz and R. Verbrugge, ‘Collective Commitments’, in: *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS’96)*, AAAI Press, Menlo Park, CA, 1996, pp. 56–63.
- [34] B. Dunin-Kępicz and R. Verbrugge, ‘Collective Motivational Attitudes in Cooperative Problem Solving’, in: *Proceedings of the 1st International Workshop of Central and Eastern Europe on Multi-Agent System (CEEMAS’99)*, V. Gorodetsky et al (eds.), 1999, p. 22–41.
- [35] B. Dunin-Kępicz and R. Verbrugge, ‘A Reconfiguration Algorithm for Distributed Problem Solving’, *Engineering Simulation* **18**, 2001, pp. 227–246.

- [36] B. Dunin-Kępcicz and R. Verbrugge, ‘Collective Intentions’, *Fundamenta Informaticae*, to be published.
- [37] E. Dubois, P. du Bois and J.M. Zeippen, ‘A Formal Requirements Engineering Model for Real-Time, Concurrent, and Distributed Systems’, in: *Proceedings of the Real-Time Systems Conference (RTS’95)*, 1995.
- [38] E.W. Dijkstra, ‘Go To Statement Considered Harmful’, *Communications of the ACM* **11**(3), 1968, pp. 147–148.
- [39] P.A.T. van Eck, *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*, PhD thesis, SIKS Dissertation Series No. 2001-8, Vrije Universiteit Amsterdam, 2001.
- [40] J. Engelfriet, C.M. Jonker and J. Treur. ‘Compositional Verification of Multi-Agent Systems in Temporal Multi-Epistemic Logic’, *Journal of Logic, Language and Information* **11**, 2002, pp. 195–225; preliminary version in: *Intelligent Agents V, Proceedings of the 5th International Workshop on Agents Theories, Architectures, and Languages (ATAL’98)*, J.P. Müller, M.P. Singh and A.S. Rao (eds.), LNCS 1555, Springer, Berlin, 1999.
- [41] R.M. van Eijk, *Programming Languages for Agent Communication*, PhD thesis, SIKS Dissertation Series No. 2000-6, Universiteit Utrecht, 2000.
- [42] A. El Fallah-Seghrouchni, S. Haddad and H. Mazouzi, ‘Protocol Engineering for Multi-agent Interaction’, in: *Multi-Agent System Engineering, Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’99)*, F.J. Garijo and M. Boman (eds.), LNAI 1647, Springer, Berlin, 1999, pp. 89–101.
- [43] J. Ferber and O. Gutknecht, ‘A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems’, *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS’98)*, IEEE Computer Society Press, 1998, pp. 128–135.
- [44] J. Ferber and O. Gutknecht, ‘Operational Semantics of a Role-based Agent Architecture’, in: *Intelligent Agents VI, Proceedings of the 6th International Workshop on Agent Theories, Architectures and Languages (ATAL’99)*, N.R. Jennings and Y. Lespérance (eds.), LNAI 1757, Springer, Berlin, 2000.

- [45] J. Ferber, O. Gutknecht, C.M. Jonker, J.P. Müller and J. Treur, ‘Organization Models and Behavioural Requirements Specification for Multi-Agent Systems’, in: *Proceedings of the 10th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’01)*, Y. Demazeau and F. Garijo (eds.), to be published.
- [46] T. Finin, D. McKay, R. Fritzson and R. McEntire, ‘KQML: An Information and Knowledge Exchange Protocol’, in: *Knowledge Building and Knowledge Sharing*, K. Fuchi and T. Yokoi (eds.), Ohmsa and IOS Press, 1994.
- [47] M. Fisher, ‘A Survey of Concurrent METATEM — the Language and its Applications’, in: *Proceedings of the 1st International Conference on Temporal Logic*, D.M. Gabbay and H.J. Ohlbach (eds.), LNAI 827, 1994, pp. 480–505.
- [48] M. Fisher and M. Wooldridge, ‘On the Formal Specification and Verification of Multi-Agent Systems’, *International Journal of Cooperative Information Systems* 6(1), special issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, 1997, pp. 37–65.
- [49] P. Gärdenfors and H. Rott, ‘Belief Revision’, in: *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, D.M. Gabbay, C.J. Hogger and J.A. Robinson (eds.), Clarendon Press, Oxford, 1995, pp. 36–132.
- [50] M.P. Georgeff and A.L. Lansky, ‘Reactive Reasoning and Planning’, in: *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI’87)*, pp. 677–682.
- [51] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, New York, 1997.
- [52] R.J. van Glabbeek and F.W. Vaandrager, ‘The Difference between Splitting in  $n$  and  $n+1$ ’, *Information and Computation* 136(2), 1997, pp. 109–142.
- [53] R.P. Goldman, K.Z. Haigh, D.J. Musliner and M. Pelican, ‘MACBeth: A Multi-Agent Constraint-Based Planner’, in: *Working Notes of the AAAI Workshop on Constraints and AI Planning*, 2000.
- [54] J.Y. Halpern and L.D. Zuck, ‘A Little Knowledge Goes a Long Way: Knowledge-Based Derivations and Correctness Proofs for a Family of Protocols’, *Journal of the ACM* 39(3), 1992, pp. 449–478.

- [55] N. Hameurlain, 'Formal Semantics for Behavioural Substitutability of Agent Components: Application to Interaction Protocols', in: *From Theory to Practice in Multi-Agent Systems, Proceedings of the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS'01)*, B. Dunin-Kęplicz and E. Nawarecki (eds.), LNCS 2296, Springer, Berlin, 2002, pp. 131–140.
- [56] P. Hendriks, N. Taatgen and T. Andringa, *Breimakers & Breinbrekers, Inleiding Cognitiewetenschap*, Addison Wesley Longman, Amsterdam, 1997, pp. 114–115, 122–123.
- [57] D.E. Herlea, C.M. Jonker, J. Treur and N.J.E. Wijngaards, 'Specification of Behavioural Requirements within Compositional Multi-Agent Systems Design', in: *Multi-Agent System Engineering, Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'99)*, F.J. Garijo and M. Boman (eds.), LNAI 1647, Springer, Berlin, 1999, pp. 8–27.
- [58] K.V. Hindriks, *Agent Programming Languages: Programming with Mental Models*, PhD thesis, SIKS Dissertation Series No. 2001-2, Universiteit Utrecht, 2001.
- [59] K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J.Ch. Meyer, 'Formal Semantics for an Abstract Agent Programming Language', in: *Intelligent Agents IV, Proceedings of the 4th International Workshop on Agent Theories, Architectures and Languages (ATAL'97)*, M. Singh, A. Rao and M. Wooldridge (eds.), LNAI 1365, Springer, Berlin, 1998, pp. 215–229.
- [60] K.V. Hindriks, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, 'Agent Programming in 3APL', *Autonomous Agents and Multi-Agent Systems* **2**, 1999, pp. 357–401.
- [61] C.A.R. Hoare, 'Communicating Sequential Processes', *Communications of the ACM* **21**(8), 1978, pp. 666–677.
- [62] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.
- [63] W. Hodges, *Model Theory*, Cambridge University Press, 1993.
- [64] D.R. Hofstadter, 'Metamagical Themas: A Coffeehouse Conversation on the Turing Test to Determine if a Machine can Think', in *Scientific American*, 1981, pp. 15–36.

- [65] Z. Huang, *Logics for Agents with Bounded Rationality*, PhD thesis, Dissertation Series DS-1994-10, Universiteit van Amsterdam, 1994.
- [66] M.N. Huhns, ‘Interaction-Oriented Programming’, in: *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE’00)*, P. Ciancarini and M.J. Wooldridge (eds.), LNCS 1957, Springer, Berlin, 2001, pp. 29–44.
- [67] N.R. Jennings, ‘Specification and Implementation of a Belief Desire Joint-Intention Architecture for Collaborative Problem Solving’, *Journal of Intelligent and Cooperative Information Systems* **2**(3), 1993, pp. 289–318.
- [68] N.R. Jennings, ‘On Agent-Based Software Engineering’, *Artificial Intelligence* **117**(2), 2000, pp. 277–296.
- [69] N.R. Jennings, K.P. Sycara and M. Wooldridge, ‘A Roadmap of Agent Research and Development’, *Journal of Autonomous Agents and Multi-Agent Systems* **1**(1), 1998, pp. 7–36.
- [70] C.M. Jonker and J. Treur, ‘Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness’, *International Journal of Cooperative Information Systems* **11**, 2002, pp. 51–92; preliminary version in: *Proceedings of the International Workshop on Compositionality (COMPOS’97)*, W.P. de Roever, H. Langmaack and A. Pnueli (eds.), LNCS 1536, Springer, Berlin, 1998, pp. 350–380.
- [71] C.M. Jonker and J. Treur, ‘A Dynamic Perspective on an Agent’s Mental States and Interaction with its Environment’, in: *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS’02)*, C. Castelfranchi and W.L. Johnson (eds.), ACM Press, 2002, pp. 865–872.
- [72] C.M. Jonker, J. Treur and W. de Vries, ‘Compositional Verification of Agents in Dynamic Environments: a Case Study’, in: *Proceedings of the KR’98 Workshop on Verification and Validation of KBS*, F. van Harmelen (ed.), 1998.
- [73] C.M. Jonker, J. Treur and W. de Vries, ‘Reuse and Abstraction in Verification: Agents Acting in Dynamic Environments’, in: *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE’00)*, P. Ciancarini and M.J. Wooldridge (eds.), LNCS 1957, Springer, Berlin, 2001, pp. 253–267.

- [74] C.M. Jonker, J. Treur and W. de Vries, ‘Reuse and Abstraction in Verification: Agents Acting in Dynamic Environments’, in: *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 7: Agent-Based Defeasible Control in Dynamic Environments, J.-J. Ch. Meyer and J. Treur (eds.), Kluwer Academic Publishers, 2002, pp. 427–454.
- [75] C.M. Jonker, J. Treur and W. de Vries, ‘Temporal Analysis of the Dynamics of Beliefs, Desires and Intentions’, *Cognitive Science Quarterly, special issue on Desires, Goals, Intentions, and Values: Computational Architectures 2*, to appear in 2002.
- [76] C.M. Jonker, J. Treur and W. de Vries, ‘Temporal Requirements for Anticipatory Reasoning about Intentional Dynamics in Social Contexts’, in: *Proceedings of the 10th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’01)*, Y. Demazeau and F. Garjo (eds.), to be published.
- [77] C.M. Jonker, J. Treur and W.C.A. Wijngaards, ‘A Modelling Environment for Mind and Matter Aspects of Intentional Behaviour’, in: *From Theory to Practice in Multi-Agent Systems, Proceedings of the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS’01)*, B. Dunin-Keplicz and E. Nawarecki (eds.), LNAI 2296, Springer, Berlin, 2002, pp. 151-160.
- [78] H. Kautz and B. Selman, ‘Blackbox: A New Approach to the Application of Theorem Proving to Problem Solving’, in: *AIPS’98 Workshop on Planning as Combinatorial Search*, 1998, pp. 58–60.
- [79] J. Kim, *Mind in a Physical world: an Essay on the Mind-Body Problem and Mental Causation*, MIT Press, Cambridge, Massachusetts, 1998.
- [80] D. Kinny, ‘Reliable Agent Computation: An Algebraic Approach’, in: *Intelligent Agents: Specification, Modeling, and Applications, Proceedings of the 4th Pacific Rim International Workshop on Multi-Agents (PRIMA’01)*, S.-T. Yuan and M. Yokoo (eds.), LNAI 2132, Springer, Berlin, 2001, pp. 31–47.
- [81] K. Konolige and M.E. Pollack, ‘Ascribing Plans to Agents: Preliminary Report’, in: *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI’89)*, Detroit, Michigan, 1989, pp. 924–930.
- [82] G. Kontonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*, John Wiley and Sons, New York, 1998.

- [83] H.J. Levesque, P.R. Cohen and J.T. Nunes, ‘On Acting Together’, in: *Proceedings of the National Conference on Artificial Intelligence*, 1990, pp. 94–99.
- [84] B. van Linder, W. van der Hoek and J.-J. Ch. Meyer, ‘Actions that Make you Change your Mind’, in: *Advances in Artificial Intelligence: Proceedings of the 19th Annual German Conference on Artificial Intelligence (KI’95)*, I. Wachsmut, C.-R. Rollinger and W. Brauer (eds.), LNAI 981, Springer, Berlin, 1995, pp. 185–196; full version in: *Knowledge and Belief in Philosophy and Artificial Intelligence*, A. Laux and H. Wansing (eds.), Akademie Verlag, Berlin, 1995, pp. 103–146.
- [85] B. van Linder, W. van der Hoek and J.-J. Ch. Meyer, ‘How to Motivate your Agents: on Making Promises that you can Keep’, in: *Intelligent Agents II, Proceedings of the 2nd International Workshop on Agent Theories, Architectures and Languages (ATAL’95)*, M.J. Wooldridge, J. Müller and M. Tambe (eds.), LNAI 1037, Springer, Berlin, 1996, pp. 17–32.
- [86] W. Łukaszewicz and E. Madalinska-Bugaj, ‘Reasoning about Action and Change: Actions with Abnormal Effects’, in: *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 7: Agent-Based Defeasible Control in Dynamic Environments, J.-J. Ch. Meyer and J. Treur (eds.), Kluwer Academic Publishers, 2002, pp. 399–409.
- [87] A. Mazurkiewicz, ‘Basic Notions of Trace Theory’, in: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J.W. de Bakker, W.-P. de Roever and G. Rozenberg (eds.), LNCS 354, Springer, Berlin, 1989, pp. 285–363.
- [88] J. McCarthy, ‘Epistemological Problems of Artificial Intelligence’, in: *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI’77)*, Morgan Kaufman Publishers, Los Altos, 1977, pp. 1038–1044.
- [89] J. McCarthy, ‘Ascribing Mental Qualities to Machines’, Technical Report, MEMO 326, AI Lab, Stanford University, 1978.
- [90] J. McCarthy and P.J. Hayes, ‘Some Philosophical Problems from the Standpoint of Artificial Intelligence’, *Machine Intelligence* **4**, 1969, pp. 463–502.
- [91] J.-J. Ch. Meyer and E.P. de Vink, ‘Step Semantics for “True” Concurrency with Recursion’, *Distributed Computing* **3**, 1989, pp. 130–145.

- [92] J.P. Müller, ‘Control Architectures for Autonomous and Interacting Agents: A Survey’, in: *Intelligent Agent Systems: Theoretical and Practical Issues*, L. Cavedon, A. Rao, W. Wobcke (eds.), LNCS 1209, Springer, Berlin, 1996, pp. 1–26.
- [93] J.P. Müller, ‘A Cooperation Model for Autonomous Agents’, in: *Intelligent Agents III, Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages (ATAL’96)*, J.P. Müller, M.J. Wooldridge and N.R. Jennings (eds.), LNAI 1193, Springer, Berlin, 1997, pp. 245–260.
- [94] R.J. Nelson, *The Logic of Mind*, 2nd edition, Kluwer Academic Publishers, Dordrecht, 1989.
- [95] A. Omicini, F. Zambonelli, M. Klusch and R. Tolksdorf (eds.), *Coordination of Internet Agents—Models, Technologies and Applications*, Springer, Berlin, 2001, Chapter 2, pp. 49–54.
- [96] D. Peleg, ‘Concurrent Dynamic Logic’, *Journal of the Association for Computing Machinery* **34**(2), 1987, pp. 450–479.
- [97] J.S. Penberthy and D.S. Weld, ‘UCPOP: A Sound, Complete, Partial Order Planner for ADL’, in: *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR’92)*, B. Nebel, C. Rich and W. Swartout (eds.), Morgan Kaufmann, 1992, pp. 103–114.
- [98] G.D. Plotkin, *A structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.
- [99] M.E. Pollack, ‘The Uses of Plans’, *Artificial Intelligence* **57**(1), 1992, pp. 43–68.
- [100] R.F. Port, T. van Gelder (eds.), *Mind as Motion: Explorations in the Dynamics of Cognition*, MIT Press, Cambridge, Massachusetts, 1995.
- [101] A.S. Rao, ‘AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language’, in: *Agents Breaking Away*, W. van der Velde and J.W. Perram (eds.), LNAI 1038, Springer, Berlin, 1995, pp. 341–354.
- [102] A.S. Rao and M.P. Georgeff, ‘Modelling Rational Agents within a BDI-Architecture’, in: *Proceedings of the 2nd International Conference*

*on Principles of Knowledge Representation and Reasoning (KR'91)*, J. Allen, R. Fikes and E. Sandewall (eds.), Morgan Kaufmann, 1991, pp. 473–484.

- [103] A.S. Rao, M.P. Georgeff and E.A. Sonenberg, ‘Social Plans: a Preliminary Report’, in: *Decentralized AI 3, Proceedings of the 3rd European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'91)*, E. Werner and Y. Demazeau (eds.), Elsevier, Amsterdam, 1992, pp. 57–76.
- [104] R. Reiter, ‘The Frame Problem in the Situation Calculus: a Simple Solution (Sometimes) and a Completeness Result for Goal Regression’, in: *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz (ed.), Academic Press, 1991, pp. 359–360.
- [105] R. Reiter, ‘Proving Properties of States in the Situation Calculus’, *Artificial Intelligence* **64**, 1993, pp. 337–351.
- [106] R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press, 2001.
- [107] S. Rosenschein and L.P. Kaelbling, ‘The Synthesis of Digital Machines with Provable Epistemic Properties’, in: *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge (TARK'86)*, J.Y. Halpern (ed.), Morgan Kaufmann, 1986, pp. 83–98.
- [108] E.D. Sacerdoti, *A Structure for Plans and Behavior*, American Elsevier, New York, 1977.
- [109] E. Sandewall, *Features and Fluents; The Representation of Knowledge about Dynamical Systems, Volume I*, Oxford University Press, 1994.
- [110] E. Sandewall and Y. Shoham, ‘Non-monotonic Temporal Reasoning’, in: *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4 on Epistemic and Temporal Reasoning, D.M. Gabbay, C.J. Hogger and J.A. Robinson (eds.), Clarendon Press, Oxford, 1995.
- [111] V.A. Saraswat, *Concurrent Constraint Programming*, The MIT Press, Cambridge, Massachusetts, 1993.
- [112] R. Sethi, *Programming Languages: Concepts & Constructs*, Addison-Wesley, 1997.

- [113] Y. Shoham, ‘Agent-Oriented Programming’, *Artificial Intelligence* **60**, 1993, pp. 51–92.
- [114] M.P. Singh, ‘Social and Psychological Commitments in Multiagent Systems’, in: *AAAI Fall Symposium on Knowledge and Action at Social and Organizational Levels*, 1991, pp. 104–106.
- [115] F. Stulp and R. Verbrugge, ‘A Knowledge-Based Algorithm for the Internet Protocol TCP’, *Bulletin of Economic Research* **54**(1), 2002, pp. 69–94.
- [116] A.S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992.
- [117] M. Tambe, ‘Teamwork in Real-world, Dynamic Environments’, in: *Proceedings of the 2nd International Conference on Multi-Agent Systems (ICMAS’96)*, AAAI Press, Menlo Park, 1996.
- [118] M. Tambe, ‘Towards Flexible Teamwork’, *Journal of Artificial Intelligence Research* **7**, 1997, 83–124.
- [119] E.P.K. Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London and San Diego, 1993, ISBN 0-12-701610-4.
- [120] A.B. Tucker (ed.), *The Computer Science and Engineering Handbook*, CRC Press, 1997, Chapters 94–98, pp. 1983–2119.
- [121] W. de Vries, ‘Motivated Machines and Longing Logics; An Account of Belief, Desire and Intention in Abstract Automata and Symbolic Logics’, Master thesis, Technical Report, INF/SCR-96-35, Universiteit Utrecht, 1996.
- [122] W. de Vries, F.S. de Boer, K.V. Hindriks, W. van der Hoek and J.-J. Ch. Meyer, ‘A Programming Language for Coordinating Group Actions’, in: *From Theory to Practice in Multi-Agent Systems, Proceedings of the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS’01)*, B. Dunin-Kępicz and E. Nawarecki (eds.), LNAI 2296, Springer, Berlin, 2002, pp. 313–321.
- [123] W. de Vries, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, ‘An Operational Real-Time Model for Interacting Agents’, in: *Electronical Proceedings of the 10th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW’01)*, Y. Demazeau and F. Gar-  
 ijo (eds.), 2001.

- [124] W. de Vries, F.S. de Boer, W. van der Hoek and J.-J. Ch. Meyer, ‘A Truly Concurrent Model for Interacting Agents’, in: *Intelligent Agents: Specification, Modeling and Applications, Proceedings of the 4th Pacific Rim International Workshop on Multi-Agents (PRIMA’01)*, S.-T. Yuan and M. Yokoo (eds.), LNAI 2132, Springer, Berlin, 2001, pp. 16–30.
- [125] G. Weiss, *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*, The MIT Press, 1999, pp. 1–9.
- [126] D.S. Weld, ‘Recent Advances in AI Planning’, *AI Magazine* **20**(2), 1999, pp. 93–123.
- [127] G. Winskel, ‘An Introduction to Event Structures’, in: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J.W. de Bakker, W.-P. de Roever and G. Rozenberg (eds.), LNCS 354, Springer, Berlin, 1989, pp. 364–397.
- [128] M. Wooldridge, ‘Agent-based Software Engineering’, *IEE Proceedings on Software Engineering* **144**(1), 1997, pp. 26–37.
- [129] M. Wooldridge and P. Ciancarini, ‘Agent-Oriented Software Engineering: The State of the Art’, in: *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE’00)*, P. Ciancarini and M.J. Wooldridge (eds.), LNCS 1957, Springer, Berlin, 2001, pp. 1–28.
- [130] M. Wooldridge and N.R. Jennings, ‘Intelligent Agents: Theory and Practice’, *The Knowledge Engineering Review* **10**(2), 1995, pp. 115–152.
- [131] M. Wooldridge and N.R. Jennings, ‘The Cooperative Problem Solving Process’, *Journal of Logic & Computation* **9**(4), 1999, pp. 563–592.
- [132] M.J. Wooldridge and A. Lomuscio, ‘Reasoning about Visibility, Perception and Knowledge’, in: *Intelligent Agents VI, Proceedings of the 6th International Workshop on Agent Theories, Architectures and Languages (ATAL’99)*, N.R. Jennings and Y. Lespérance (eds.), LNAI 1757, Springer, Berlin, 2000, pp. 1–12.
- [133] FIPA specification repository, URL:  
<http://www.fipa.org/repository/index.html>.

---

## Samenvatting

---

Computers en hun toepassingen worden steeds ingewikkelder. Moderne computersystemen bestaan vaak uit min of meer zelfstandige onderdelen (hardware en software) die interacteren met hun omgeving. Computers communiceren met andere computers, ontvangen informatie en opdrachten van hun menselijke gebruikers en kunnen sensoren hebben om hun fysieke of virtuele omgeving waar te nemen. Deze toegenomen interactie vraagt om handvaten om de inherent grotere complexiteit te kunnen beheersen.

Een belangrijk middel om ingewikkelde zaken beter hanteerbaar te maken is abstractie. Dit betekent dat je intuïtieve concepten zoekt, deze vanuit een duidelijke visie op de materie benoemt en details weglaat. Een goed voorbeeld van abstractie is de metrokaart van Parijs: hoewel Parijs een grote stad is met een ingewikkelde plattegrond, is het met een abstracte metrokaart snel duidelijk hoe je moet reizen, en welke kleuren metrolijnen (de intuïtieve concepten) je moet nemen. In het onderzoeksgebied van multi-agentsystemen, waarin het onderzoek in dit proefschrift past, wordt veel gebruik gemaakt van abstractie. Bij deze nieuwe visie op software zijn de basisconcepten vertrouwde menselijke begrippen zoals kennis, geloof, plannen, doelen, acties, intenties en observaties. Een agent is een stuk software met een zekere mate van zelfstandigheid, ontworpen en/of gebouwd met behulp van genoemde concepten, dat interacteert met andere agenten en met zijn omgeving, op zo'n manier dat de agent inspeelt op de voortdurende dynamiek van de omgeving en z'n doelstellingen behaalt.

Er is al veel onderzoek verricht naar de ‘losse’ agent. In mijn proefschrift richten we ons dan ook op de interactie tussen agenten onderling en hun omgeving. We introduceren een aantal nieuwe abstracte concepten voor agentinteractie, vanuit verschillende gezichtspunten binnen het onderzoeksgebied van de multi-agent systemen. Een gevaar van het gebruik van handige abstracte concepten is dat ze geen enkele of een erg vage relatie hebben met de onderliggende complexe werkelijkheid van de berekeningsprocessen in computers. Om te voorkomen dat we luchtkastelen bouwen, zorgen we ervoor dat elk nieuw abstract concept verankerd is in de complexe computationele werkelijkheid, door een formele definitie te geven die een relatie legt tussen het concept en zijn computationele betekenis. Op deze manier zijn nieuwe abstracte interactieconcepten niet alleen intuïtief en handig, maar ook krachtig en correct.

We beginnen in hoofdstuk 2 met het verankeren van drie al bestaande en veel gebruikte agentconcepten, te weten geloof, verlangen en intentie, in het uiterlijke gedrag van de agent. Dit gedrag bestaat uit het observeren van de omgeving, het communiceren met andere agenten en het doen van acties. De drie abstracte concepten geloof, verlangen en intentie worden vaak gebruikt om de interne toestand van een agent in uit te drukken. Een agent gelooft bepaalde informatie (omdat de agent observaties gedaan heeft, bijvoorbeeld), hij verlangt een bepaalde toestand te bereiken en om dat te realiseren heeft hij de intentie om zekere acties uit te voeren. Omdat de interne, mentale toestand van een agent enkel afhangt van diens interactie met de omgeving in de loop van de computationele geschiedenis is het mogelijk om een relatie te leggen tussen intenties, verlangens en geloofde feiten en de voorafgaande interactiegeschiedenis. In hoofdstuk 2 geven we formele criteria die deze relatie vastleggen. Met behulp van deze criteria is het ondermeer mogelijk dat één agent het uiterlijke gedrag van een andere agent observeert, en vervolgens geloof in een feit, verlangen naar een toestand of intentie tot het doen van een actie toeschrijft aan de geobserveerde agent, met het doel het geobserveerde gedrag te verklaren en te voorspellen, en wellicht ook te manipuleren.

Het volgende hoofdstuk, hoofdstuk 3, ligt op het gebied van verificatie van agent systemen, dat wil zeggen het bewijzen dat agenten zich zo gedragen als de bedoeling is. Omdat multi-agent systemen complex gedrag vertonen is verificatie vaak de enige manier om vast te stellen dat een systeem goed werkt. Maar voor verificatie is een wiskundige taal met bijbehorende bewijsprincipes nodig, een logica. Systeem ontwikkelaars zijn in het algemeen niet erg bewaam in logica, en dus wordt er in de praktijk zelden geverifieerd. Om het verificatieproces te vergemakkelijken introduceren we twee principes, namelijk taalabstractie en generieke herbruikbare stelsels van eigenschappen en bewijzen. Taalabstractie maakt de logische formules begrijpelijker, zodat ook de

a-logicus ze kan begrijpen. Door bibliotheken te bouwen met daarin herbruikbare stellingen en hun bewijzen wordt het verifiëren van een bepaald systeem ook makkelijker; je zoekt een bewijs op, instantieert de systeemspecifieke parameters, en kijkt of de condities waaronder het bewijs gemaakt is voor dit bepaalde systeem gelden. Dit laatste kan weer tot nieuwe verificatiestappen leiden. In hoofdstuk 3 illustreren we dit idee met een herbruikbaar bewijs van de eigenschap dat acties van een individuele agent in een dynamische omgeving slagen in bepaalde omstandigheden.

In hoofdstuk 4 presenteren we een nieuw model van agenten, waarin we ons vooral richten op de interactie van agenten met elkaar en met hun omgeving. Dit betekent dat we de dynamische omgeving niet weglaten uit het model, zoals in andere modellen wel vaak gebeurt. Elke actie heeft een tijdsduur, en acties die gedeeltelijk gelijktijdig plaatsvinden kunnen met elkaar interfereren en tot ongewenste effecten leiden, of juist tot een positiever resultaat leiden (bijvoorbeeld: een grote tafel is te verplaatsen door met z'n tweeën elk een kant op te tillen, terwijl het in je eentje niet gaat). Het model bestaat uit een flexibele programmeertaal en haar betekenis. Die betekenis geven we met behulp van een formele semantiek, waarmee voor elk nieuw abstract concept in de programmeertaal met wiskundige precisie een betekenis wordt vastgelegd. Een voorbeeld van een nieuw abstract interactieconcept in ons model is de groepsactie; agenten voeren samen een groepsactie uit door elk een individuele actie te doen, op een goed gesynchroniseerde manier (zoals bij het samen optillen van een tafel).

In hoofdstuk 5 is de groepsactie het centrale concept. We ontwikkelen een programmeertaal, genaamd GrAPL (Group Agent Programming Language), die specifiek bedoeld is om multi-agent systemen mee te programmeren waarin de agenten groepen kunnen vormen om gezamenlijk acties te doen. Voordat een groep daadwerkelijk een groepsactie uitvoert communiceren agenten met elkaar over de details van de actie en de samenstelling van de groep agenten die de actie gaat doen. Ze stellen eisen aan de groepsactie in de vorm van logische formules. Als de eisen van de agenten verenigbaar zijn, dan kan de actie uitgevoerd worden. Zowel voor de groepscommunicatie als voor het uitvoeren van groepsacties hebben we nieuwe programmeerconcepten gevonden. Net als in het vorige hoofdstuk zorgt de formele semantiek van de programmeertaal ervoor dat de abstracte concepten een eenduidig gedefiniëerde betekenis hebben.

We kunnen het idee van hoofdstuk 5 generaliseren door te kijken naar groepsplannen in plaats van naar losse groepsacties. Een groepsplan is een geheel van individuele acties en groepsacties, en hun verbanden in de tijd (gebeuren acties tegelijk of na elkaar?). Elk plan is gemaakt met een doel voor ogen; als het plan goed in elkaar zit, en de omstandigheden meewerken, dan

wordt dit doel bereikt als het plan uitgevoerd wordt. In hoofdstuk 6 introduceren we een coördinatietaal voor het maken en uitvoeren van plannen voor en door groepen agenten. Een coördinatietaal is een taal van een hoog abstractieniveau die gebruikt kan worden om agenten die in verschillende programmeertalen gebouwd zijn met elkaar te laten interacteren. Onze coördinatietaal bevat communicatieconstructen, zodat groepen agenten kunnen overleggen over een plan in wording, kunnen besluiten dat het plan voldoende uitgewerkt is en het plan in werking kunnen zetten. Ook deze taal is uitgerust met een formele semantiek om de nieuwe abstracte interactieconcepten grondig te verankeren.

In dit proefschrift hebben we manieren gevonden om de grote complexiteit die agentinteractie met zich meebrengt te structureren met behulp van intuïtieve abstracte concepten. De concepten zijn zo intuïtief omdat ze geïnspireerd zijn door begrippen die we dagelijks gebruiken om het menselijk handelen en redeneren te beschrijven. We hebben er altijd zorg voor gedragen dat de abstracte concepten een duidelijke betekenis hebben in termen van de onderliggende complexe details. We introduceren dus geen lege frases, maar goed gedefiniëerde, krachtige begrippen. Hoewel het werk in dit proefschrift theoretisch van aard is, is het van waarde voor de praktijk, omdat het gebruik van goed gefundeerde, intuïtieve abstracte concepten het ontwerpen, bouwen en analyseren van de computersystemen die we nodig hebben mogelijk maakt.

---

# Curriculum Vitae

---

Wietske de Vries

**12 december 1971:** Geboren te Sneek

**1984 – 1990:** Voorbereidend Wetenschappelijk Onderwijs aan het Christelijk College Nassau Veluwe te Harderwijk

**1990 – 1996:** Studie Informatica aan de Universiteit Utrecht

**1996 – 1999:** Assistent in Opleiding bij de afdeling Kunstmatige Intelligentie, faculteit Exacte Wetenschappen, Vrije Universiteit Amsterdam

**2000 – 2002:** Assistent in Opleiding aan het Instituut voor Informatica en Informatiekunde, faculteit Wiskunde en Informatica, Universiteit Utrecht



---

## SIKS Dissertation Series

---

- 1998-1 Johan van den Akker (CWI)  
DEGAS - An Active, Temporal Database of Autonomous Objects
- 1998-2 Floris Wiesman (UM)  
Information Retrieval by Graphically Browsing Meta-Information
- 1998-3 Ans Steuten (TUD)  
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
- 1998-4 Dennis Breuker (UM)  
Memory versus Search in Games
- 1998-5 E.W. Oskamp (RUL)  
Computerondersteuning bij Straftoemeting
- 1999-1 Mark Sloof (VU)  
Physiology of Quality Change Modelling; Automated Modelling of Quality Change of Agricultural Products
- 1999-2 Rob Potharst (EUR)  
Classification using Decision Trees and Neural Nets
- 1999-3 Don Beal (UM)  
The Nature of Minimax Search
- 1999-4 Jacques Penders (UM)  
The Practical Art of Moving Physical Objects

- 1999-5 Aldo de Moor (KUB)  
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
- 1999-6 Niek J.E. Wijngaards (VU)  
Re-design of Compositional Systems
- 1999-7 David Spelt (UT)  
Verification Support for Object Database Design
- 1999-8 Jacques H.J. Lenting (UM)  
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.
- 2000-1 Frank Niessink (VU)  
Perspectives on Improving Software Maintenance
- 2000-2 Koen Holtman (TUE)  
Prototyping of CMS Storage Management
- 2000-3 Carolien M.T. Metselaar (UvA)  
Sociaal-Organisatorische Gevolgen van Kennistechnologie; een Procesbenadering en Actorperspectief.
- 2000-4 Geert de Haan (VU)  
ETAG, A Formal Model of Competence Knowledge for User Interface Design
- 2000-5 Ruud van der Pol (UM)  
Knowledge-based Query Formulation in Information Retrieval
- 2000-6 Rogier van Eijk (UU)  
Programming Languages for Agent Communication
- 2000-7 Niels Peek (UU)  
Decision-theoretic Planning of Clinical Patient Management
- 2000-8 Veerle Coup (EUR)  
Sensitivity Analysis of Decision-Theoretic Networks
- 2000-9 Florian Waas (CWI)  
Principles of Probabilistic Query Optimization
- 2000-10 Niels Nes (CWI)  
Image Database Management System Design Considerations, Algorithms and Architecture
- 2000-11 Jonas Karlsson (CWI)  
Scalable Distributed Data Structures for Database Management
- 2001-1 Silja Renooij (UU)  
Qualitative Approaches to Quantifying Probabilistic Networks
- 2001-2 Koen Hindriks (UU)  
Agent Programming Languages: Programming with Mental Models

- 2001-3 Maarten van Someren (UvA)  
Learning as Problem Solving
- 2001-4 Evgueni Smirnov (UM)  
Conjunctive and Disjunctive Version Spaces with Instance-Based  
Boundary Sets
- 2001-5 Jacco van Ossenbruggen (VU)  
Processing Structured Hypermedia: A Matter of Style
- 2001-6 Martijn van Welie (VU)  
Task-based User Interface Design
- 2001-7 Bastiaan Schonhage (VU)  
Diva: Architectural Perspectives on Information Visualization
- 2001-8 Pascal van Eck (VU)  
A Compositional Semantic Structure for Multi-Agent Systems  
Dynamics
- 2001-9 Pieter Jan 't Hoen (RUL)  
Towards Distributed Development of Large Object-Oriented  
Models, Views of Packages as Classes
- 2001-10 Maarten Sierhuis (UvA)  
Modeling and Simulating Work Practice BRAHMS: a Multiagent  
Modeling and Simulation Language for Work Practice Analysis and  
Design
- 2001-11 Tom M. van Engers (VU)  
Knowledge Management: The Role of Mental Models in Business  
Systems Design
- 2002-01 Nico Lassing (VU)  
Architecture-Level Modifiability Analysis
- 2002-02 Roelof van Zwol (UT)  
Modelling and Searching Web-based Document Collections
- 2002-03 Henk Ernst Blok (UT)  
Database Optimization Aspects for Information Retrieval
- 2002-04 Juan Roberto Castelo Valdueza (UU)  
The Discrete Acyclic Digraph Markov Model in Data Mining
- 2002-05 Radu Serban (VU)  
The Private Cyberspace Modeling Electronic Environments  
Inhabited by Privacy-concerned Agents
- 2002-06 Laurens Mommers (UL)  
Applied Legal Epistemology; Building a Knowledge-based  
Ontology of the Legal Domain

- 2002-07 Peter Boncz (CWI)  
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications
- 2002-08 Jaap Gordijn (VU)  
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas
- 2002-09 Willem-Jan van den Heuvel (KUB)  
Integrating Modern Business Applications with Objectified Legacy Systems
- 2002-10 Brian Sheppard (UM)  
Towards Perfect Play of Scrabble
- 2002-11 Wouter C.A. Wijngaards (VU)  
Agent Based Modelling of Dynamics: Biological and Organisational Applications
- 2002-12 Albrecht Schmidt (UvA)  
Processing XML in Database Systems
- 2002-13 Hongjing Wu (TUE)  
A Reference Architecture for Adaptive Hypermedia Applications





*Take me to the place  
where your heart hurts most  
Lead me through the dark world gates  
down there, where all the ghosts  
of sorrow and pain, fear and despair stay hiding  
And we'll walk right through, to our own way, our own place.*

*There's a beach that we'll walk,  
So long and so broad  
Oceans await, miles longer than pain  
In my glad dreams I take you there and it's easy  
'cause the work and the hours and the pain  
are far behind our sure steps.*

*My heart longs to be next to you  
My heart wants to be there, be there with you  
Where it's warm and tender  
Mercy flows like a river  
And there you stand with your wide open hands and say  
Abide with me.*

*Deacon Blue*